Olivier Boissier
Rafael H. Bordini
Jomi F. Hübner
Alessandro Ricci

Multi-Agent Oriented Programming

Programming Multi-Agent Systems Using JaCaMo

Multi-Agent Oriented Programming

Multi-Agent Oriented Programming

Programming Multi-Agent Systems Using JaCaMo

Olivier Boissier, Rafael H. Bordini, Jomi F. Hübner, Alessandro Ricci

The MIT Press
Cambridge, Massachusetts
London, England

This book was set in Garamond by the authors using LaTeX.

Library of Congress Cataloging-in-Publication Data

Names: Boissier, Olivier, author. | Bordini, Rafael H., author. | Hübner, Jomi Fred, author. | Ricci, Alessandro, 1984- author.
Title: Programming collective autonomous systems : multi-agent oriented programming using JaCaMo / Olivier Boissier, Rafael H. Bordini, Jomi F. Hübner, and Alessandro Ricci.
Description: Cambridge, Massachusetts : The MIT Press, [2020] | Series: Intelligent robotics and autonomous agents series | Includes bibliographical references and index.
Identifiers: LCCN 2019059143 | ISBN 9780262044578 (hardcover)
Subjects: LCSH: Multiagent systems. | Intelligent agents (Computer software)
Classification: LCC QA76.76.I58 B67 2020 | DDC 006.3/0285436–dc23
LC record available at https://lccn.loc.gov/2019059143

To Marie Dominique, Noémie, Mathilde, and Guillemette
— OB

To Ismael, Kyra, Thor, and Maggi
— RB

To Ilze, Morgana, and Thales
— JH

To Sara, Sofia, Leonardo, Eugenio, and Gregorio
— AR

Contents

List of Figures

List of Research Corners

List of Technology Corners

Acknowledgments

We dedicated much time to the research, development, and writing leading to this book, particularly in the last few years. However, this book would not have been possible without the help and inspiration from many people whom we acknowledge individually here. Before we do so, we would like to make some joint acknowledgments.

The collaboration undertaken to define the Multi-Agent Oriented Programming (MAOP) approach and JaCaMo platform would not have been possible without the Dagstuhl Seminars. We would like to thank the organizers and the people from Schloss Dagstuhl for providing such a stimulating environment, which creates opportunities for sharing knowledge and initiating fruitful collaborations.

We would also like to thank the Autonomous Agents and Multi-Agent Systems (AAMAS) community for feedback, discussions during the conferences, workshops, and the tutorials we had the pleasure to give at various venues. We thank in particular the European Agent Systems Summer School (EASSS) and the Workshop and School on Agents, Environments, and Applications (WESAAC) participants and organizers for the tutorials we had the pleasure to give, which were all opportunities to discuss and share ideas on MAOP. We thank the users and contributors of Jason, CArtAgO, *M*OISE, and JaCaMo for feedback, bug fixes, and ideas developed under the principles of free and open-source software.

OB would like to thank Mines Saint-Etienne, including its master and PhD program in computer science. He is very grateful to many people for the always stimulating and never-ending discussions and collaborations, including Yves Demazeau, Jaime Simão Sichman, Cristiano Castelfranchi, Antonio Carlos da Rocha Costa, and the French research community on artificial intelligence and multi-agent systems. He would like to thank the members of the Computer Science and Intelligent Systems department and of the Connected Intelligence team; and the collaborators who participated directly or indirectly to the discussions; and to the developments of the ideas, models, and tools for the engineering of decentralized and open intelligent systems, in particular Fabien Badeig, Flavien Balbo, Matteo Baldoni, Cristina Baroglio, Philippe Beaune, Carlos Carrascosas, Agnes Crepet, Adina Florea, Catherine Garbay, Maxime Lefrançois, Roberto Micalizio, Gauthier Picard, Fano Ramparany, Claudette Sayettat, Laurent Vercouter, and Antoine Zimmermann. He is very grateful to the master and PhD students he had the pleasure to advise and/or to collaborate with. In particular, he would like to thank those who contributed directly or indirectly to the different MAOP dimensions: Mohamad Allouche, Mahdi Hannoun, Hubert Proton, Thibault Caron, Mihnea Bratu, Cosmin Carabelea, Oana Bucur, Benjamin Gateau, Grégoire Danoy, Luciano Coutinho, Camille Person, Andrei Ciortea, Alessandru Sorici, Réda Yaich, Amro Najjar, Andrea Santi, Michele Piunti, Maiquel De Brito, Maicon Zatelli, Daniella Maria Uez, Rosine Kitio, Lauren Thevin, Nicolas Cointe, Katherine May, Stefano Tedeschi, and Iago Felipe Trentin. He also wishes to acknowledge the support of the UFSC in Brazil for the visiting academic position; the USP COFECUB 98-04 and International Cooperation and Mobility Rhône-Alpes (CMIRA) 2012 Région Rhône Alpes; the ForTrust and ETHICs and Autonomous Agents (ETHICAA) Agents Autonomes et Ethique (ANR) projects; and Orange Labs for enabling the funding of visits, post-docs; and PhD students that contributed directly or indirectly to parts of the MAOP work presented in the book.

RB would like to thank the University of Liverpool, the University of Durham, the Universidade Federal do Rio Grande do Sul (UFRGS), and Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS), including the postgraduate program in computer science), institutions with which he was affiliated throughout the period of development of Jason, later leading to JaCaMo. He also would like to thank the University of Genoa and the University of Oxford for the visiting academic positions over his sabbatical period, during which the book was finalized. The work in this book is based on Jason, which is a joint work with JH and has been a source of dedication throughout his career. Although this research career was influenced by a large number of people, he would like to first of all thank Antonio Carlos Rocha Costa for being a source of inspiration not only to him but to the entire AI community in Brazil, where he was a pioneer of AI research, and for being a role model for academics in his ethical and dedicated commitment to high standards in scientific research and teaching. RB is also very grateful to his PhD supervisor, Professor John A. Campbell (*in memoriam*), and all his collaborators over the years as well as colleagues from the AAMAS and more particularly the engineering multi-agent systems (EMAS) research communities, including Renata Vieira, Antonio Carlos da Rocha Costa, Alvaro Moreira, Viviana Mascardi, Davide Ancona, Michael Wooldridge, Michael Fisher, Willem Visser, Brian Logan, Natasha Alechina, Jürgen Dix, Amal El Fallah Seghrouchni, Peter McBurney, Simon Parsons, Louise Dennis, Berndt Farwer, Ana Bazzan, Luis Silva, Diana Adamatti, João Leite, Paolo Torroni, Lars Braubach, Alexander Pokahr, Moser Fagundes, Sandro Fiorini, Marcia Campos, Dejan Mitrovic, Stefan Sarkadi, Yves Lesperance, Munindar Singh, Jørgen Villadsen, Francisco Grimaldo, Wojtek Jamroga, Mateo Baldoni, Cristina Baroglio, Rem Collier, Sebastian Sardina, and many others. He is very grateful to all the students he has supervised over the years, who contributed immensely to his research, and in particular, recently to the SMART research group, including Fabio Okuyama, Rodrigo Machado, Patricia Shaw, Thomas Klapiscak, Rafael Cardoso, Alison Panisson, Tulio Basegio, Giovani Farias, Tabajara Krausburg, Debora Engelmann, João Brezolin, Artur Freitas, Daniela Schmidt, Lucas Hilgert, Alexandre Zamberlan, Vagner Gabriel, Juliana Damasio, and Victor Melo. Finally, he would like to thank the National Council of Scientific and Technological Development (CNPq), the Brazilian Federal Agency for Support and Evaluation of Graduate Education (CAPES), the Fundacao de Amparo a Pesquisa do Estado do Rio Grande do Sul (FAPERGS), and Samsung Brazil for funding research projects that contributed directly or indirectly to parts of the work presented in this book.

JH would like to thank the Federal University of Santa Catarina (UFSC) and Regional University of Blumenau (FURB) institutions with which he was affiliated throughout the period of development of Moise, Jason, and JaCaMo. He also would like to thank Mines Saint-Etienne for several visiting periods and sabbatical years during which the book was born. During his academic pursuit, he has been fortunate to be advised by two distinguished researchers who left a fundamental impact on his work on MAS, Antonio Carlos da Rocha Costa and Jaime Simão Sichman. He is also very grateful to all the students he supervised over the years who contributed immensely to his research. In particular, he wants to thank his PhD students Maiquel De Brito, Maicon Zatelli, Cleber Amaral, Daniella Uez, Tiago Schmitz, and Gustavo Ortiz-Hernández, who directly contributed to the JaCaMo project. Finally, he would like to thank CNPq, CAPES, and Petrobras for funding research projects that contributed directly or indirectly to parts of the work presented in this book.

AR would like to thank Alma Mater Studiorum Università di Bologna, the Department of Computer Science and Engineering (DISI), and in particular the Cesena Campus, whom AR had the chance to join and grow up with thanks to a great research group composed of Antonio Natali, Andrea Omicini, Enrico Denti, and Mirko Viroli. Every contribution of AR in this book is the result of joyful, tough, and never-ending discussions and reflections with these guys, mentors, and friends. AR is very grateful too to Rune Gustuvsson, Martin Fredriksson, and the SOCLAB at the Blekinge Institute of Technology (BTH) in Sweden, where AR spent insightful months during his PhD studies a long time ago and "where artifacts come from."

AR is grateful to the PhD students whom he had the chance to supervise and grow up with and who eventually gave important contributions and ideas on the topic of this book: Michele Piunti, Andrea Santi, and Angelo Croatti. Finally AR is deeply grateful to all the inspiring researchers whom he met and worked with and who contributed either directly or indirectly to this book in these years: Cristiano Castefranchi, Luca Tummolini, Carlos Carrascosas, Rem Collier, Andrei Ciortea, Viviana Mascardi, Danny Weyns, Fabien Michel, Amal El Fallah Seghrouchni, Matteo Baldoni, Cristina Baroglio, Giovanni Rimassa, Giuseppe Vizzari, Maicon Rafael Zateli, Xavier Limón, Daghan L. Acay, Ambra Molesini, Stefano Mariani, Franco Zambonelli, and Marco Mamei.

Finally, we would like to thank our families, who patiently accepted the many meetings and work sessions during the evenings, weekends, and holidays.

O. Boissier, R. H. Bordini, J. F. Hübner, A. Ricci

February 2020

1 Introduction

Modern software applications have to deal with an increasing level of autonomy of interconnected software systems, and above all with the integration of countless systems that are not known in advance. Current trends such as smart cities, intelligent transportation systems, and industry fostered by the development of IoT (Internet of Things), for instance, point to even more complex scenarios in which adaptive and open teams of intelligent autonomous software entities and robots will interact with humans and everyday objects, all interconnected. *Multi-agent systems* (MAS) can be used as a suitable paradigm for modeling and engineering such systems. A multi-agent system is an organized ensemble of autonomous goal-oriented entities called *agents*, communicating with each other and interacting within an *environment*. At the individual level, each agent can have its own goals and tasks to pursue autonomously by deciding what actions to do. As an ensemble, agents typically need to coordinate and cooperate in order to achieve the global objective of the MAS as a whole, as an *organization*. This book is about *programming* multi-agent systems, using an integrated approach that we refer to as *multi-agent oriented programming* (MAOP).

multi-agent system

multi-agent oriented programming

In the literature, many relevant techniques for dealing with multi-agent systems emerged in different contexts—the main examples are artificial intelligence (AI), distributed AI, software engineering (SE), simulation—and some of them led to concrete programming models for dealing with the increasing levels of autonomy and complexity of interactions in modern systems. In this direction, MAOP provides a structured approach based on three interrelated sets of concepts and programming abstractions (hereafter called *dimensions*) that are useful for designing such complex systems: the *agent dimension* that is used to program the individual (interacting) autonomous entities; the *environment dimension* that is used to program the shared resources and means used by agents to work, interact, and connect to the real world; and the *organization dimension* used to structure and regulate the complex interrelations taking place between the autonomous agents in the shared environment.

JaCaMo In order to see in practice MAOP concepts and methods, we use the *JaCaMo* platform, which is an open-source MAS technology supporting the integration between the three dimensions that we consider in this book.

1.1 Objectives

This book aims at providing the reader with the *know-why* and *know-how* necessary to master MAOP.

- The know-why knowledge brings to the reader a deep understanding of the MAOP foundations as well as a flavor of the benefits and limitations of such an approach for programming multi-agent systems.
- The know-how knowledge provides the reader with the ability to use MAOP for developing multi-agent systems.

An MAOP approach involves various dimensions of concepts that bring various benefits. However, mastering the use of all the dimensions and their specific concepts is a long-term task that needs practicing through the development of various applications. Throughout this book, the reader learns some practical ability to use MAOP techniques and programming patterns originating from the experience that the authors of this book acquired while developing applications over the years.

1.2 Challenges

In general, multi-agent systems are used to model, design, and program complex systems in various application domains. The modeling and engineering features of MAOP that we introduce in this book are well adapted to address challenging features or properties related to the complexity of such systems.

autonomy The fundamental feature of the systems we address in this book is *autonomy*. Autonomy is increasingly required everywhere in computer science, not the least in robotics, an area to which much effort of the AI and SE research communities is currently dedicated, but more generally in many other areas such as health care and smart homes. The concept of autonomy can have different interpretations depending on the specific context, and different *levels* of autonomy may be identified. In the context of this book we consider the meaning typically assumed in an AI context, that is, the property of a system's embedding and enacting some *decision making* in order to perform tasks for which the system has been designed, typically requiring interaction with some environment and adaptation to it. In order to do its job, an autonomous system must be able to act without human intervention. Nevertheless, that job could involve assisting and interacting with human users. In the context of

MAS, a specific first-class abstraction—the *agent*—is introduced to directly address this feature: agents represent entities featuring *autonomous behavior*.

Typically, the complex systems addressed in this book cannot be designed as a *single* entity centralizing all the decision making. This could result from different factors, such as the impossibility or impracticability of conveying in a single point all the information needed for the decision making, or the impossibility of processing them in an effective way using a single decision maker, that is, the need to apply different decisions concurrently in different parts of a distributed environment. Therefore, these systems need to adopt a full *decentralization* of both data and control, that is, to have multiple *loci* of control and decision making, each one dealing with a portion of the whole environment and problem. Proper *coordination* strategies are then needed to manage the dependencies among the multiple decision makers in order to achieve the objectives of the system as a whole. In addition to being decentralized, these systems can be *distributed* over multiple computing hosts and devices, which do not share memory, and typically communicate through the internet. The notion of a multi-agent system has been introduced to effectively model systems as collections of decentralized, loosely coupled autonomous entities, of which communication, coordination, and cooperation are first-class aspects.

decentralization

coordination

Distribution introduces further challenges, in terms of availability and resiliency, as well as heterogeneity, security, and openness. Regarding openness, as stated in Hewitt and De Jong (1984), "open systems are distributed, highly parallel, incrementally evolving, computer systems that are in continuous operation always capable of further growth." *Openness* thus concerns the fact that the set of elements taking part in the system is highly dynamic (i.e., the participants, whether humans or software entities, may enter and leave the system while it runs), as well as the lack of control at design time on the number and behavior of these elements. Instead, because of *heterogeneity*, different components of the system can have different characteristics and capabilities, and can run on heterogeneous hardware and software stacks. These aspects are directly captured in MAS because agents are by definition *loosely coupled*, and their communication models (e.g., speech acts based on high-level agent communication languages) do not make any assumption on the specific software stack on which they are based. The use of standard protocols and technologies (e.g., web technology) eases the interoperability of MAS technologies with mainstream and legacy technologies.

distribution

openness

heterogeneity

Finally, the progress made in AI techniques such as machine learning in recent years calls for systematic and robust approaches to embed them in the engineering of autonomous software systems, so as to increase their flexibility and, ultimately, autonomy. Flexibility and *adaptation* are required especially when we

adaptation

consider systems that are situated in dynamic and unpredictable environments, so that it is not possible (or feasible, or even convenient) to have a full model of their structure and dynamics at design time. This occurs even when we consider long-running systems situated in contexts that evolve over time. In all these cases, it is not possible (or feasible) for designers and developers to predefine at design time the full business logic to be executed at runtime by the system in order to autonomously perform tasks and achieve its goals. In AI, *planning* and *reinforcement learning* are two broad examples of techniques that can be exploited to build flexible and adaptable agents, to achieve full autonomy even in the case of unpredictable and evolving environments. Nevertheless, if a computer-based system is truly autonomous, it will be trusted only if it can explain rationally every single decision it makes. That is, *explainability* becomes a main concern, and just like autonomy, it can be defined at different levels, spanning from designers to programmers, and users. In that perspective, agents and MAS provide an architectural blueprint to integrate AI techniques in a disciplined way.

explainability

In the next chapter, we look into how the characteristics of MAOP allow developers to address all these challenges.

1.3 Approach

This book follows and combines various techniques for learning how to program multi-agent systems using MAOP:

- **Full immersion** Readers learn how to use an MAOP approach from the first program in the book, so that the core concepts of MAOP are emphasized and explained from the very beginning.
- **Incremental approach** Concepts and techniques are introduced incrementally starting from the simplest multi-agent programs, interleaving engineering principles with programming.
- **Examples** The book supports learning by experience so that the reader acquires the mechanics of MAOP, a personal set of schemes/patterns/idioms that can possibly be reused in the development of the reader's own multi-agent programs.
- **Project based** In addition to examples throughout the text, some chapters are devoted to complete case studies, which support project-based teaching.
- **Scientific foundations** The principles of MAOP are introduced with a presentation of the *why*, *what*, and *how* questions at the core of MAOP.

The JaCaMo platform underpins all these aspects, forming a general and uniform basis for the reader to develop MAOP programs. JaCaMo is the freely available, open-source development platform used throughout the book to implement the exercises, case studies, and example systems.

1.4 Intended Readership

The book is targeted to a wide readership, which includes:

- **Students** Both undergraduate and postgraduate students can use this book to learn multi-agent oriented programming in order to develop multi-agent systems. The book can be used in courses on multi-agent systems, artificial intelligence, and software engineering that present the principles of intelligent agents and multi-agent systems, or, more generally, the design of collective autonomous applications, as required in most current views on computing trends, providing new material to support practical coursework.
- **Practitioners in the area of computer programming** Technologists, advanced developers, software architects, and other practitioners can acquire a deep understanding of the concepts and foundations underlying the MAOP approach and the best practices of that approach for developing complex, open, autonomous systems. The book can help practitioners leverage their practical experience to program multi-agent systems applications in a principled way.
- **Researchers** Whether active or not in each of the various areas of research related to multi-agent systems, researchers may wish to get an overall and structured view on practical MAOP, even if they are already deeply knowledgeable in some aspects of the subject.

No specific prerequisite knowledge is necessary to read this book, in the sense that any computer scientist who knows about programming and programming languages at the level of a typical undergraduate computer science curriculum should be able to follow the book material. Although it is not mandatory, background knowledge in one of the following domains will help the reader gain a better perspective on the book material: logic programming, programming languages, object-oriented programming, multi-agent systems, and artificial intelligence.

1.5 Book Structure and Reading Guide

The remainder of this book is organized as follows. First, we set the stage with an overview of the three core conceptual dimensions of MAOP (chapter 2); then, by getting started with the platform, we show how to use MAOP in practice (chapter 3). We then go into details of the concepts and programming abstractions used at the agent (chapter 4) and environment (chapter 5) dimensions. A first case study is presented in chapter 6, with running code inspired by a smart room scenario; the system has only agents and an environment model. In chapter 7 we revisit that system, focusing on the interactions of autonomous agents. At this point, we have not yet entered the important dimension of agent *organizations*, which is fundamental for complex multi-agent systems. In chapter 8, we discuss the concepts at the organization dimension, and then in chapter 9 we extend the smart room scenario to provide a case study in which the main elements of our approach to MAOP are used together, as they would be in the development of complex MAS. Chapter 10 includes a general discussion on how MAOP can be used to integrate different technologies and to serve as the engineering approach to integrate and build intelligent systems. Finally, chapter 11 includes a selection of advanced features in MAOP and JaCaMo, showing some further directions of research involving advanced programming techniques often related to ongoing extensions to our approach, of interest mainly to advanced JaCaMo programmers and researchers.

Reading the whole book is not necessary to get some knowledge about MAOP or indeed to learn how to program with JaCaMo. As a guideline, consider reading only chapters 2, 4, 5, and 8 if you want to gain a feel for the powerful abstractions used in MAOP and also acquire a deep understanding of the related concepts and examine the matching code examples. If, on the other hand, you are interested only in practical programming rather than the foundations, you could read only chapters 3, 6, 7, and 9. Chapters 10 and 11 are useful mainly for the readers interested in *advanced* practical multi-agent oriented programming with JaCaMo. Note that the conceptual part on organizations (chapter 8) comes only near the end, as we follow an incremental approach to the presentation of MAOP in this book. We can thus present a complete system example (having only agents and an environment) in chapter 6, allowing the reader who aims to read the entire book to gain some hands-on experience as early as possible.

Throughout the book you will find two types of boxes that clearly separate from the main text flow aspects that will interest only some of the intended readership of the book. A *research corner* box will typically interest MAS researchers only, and a *technology corner* typically contains programming details that will interest only advanced programmers.

Research Corner Example

This is an example of a research corner.

Technology Corner Example

This is an example of a technology corner.

To help the reader better understand the concepts presented in the book, various figures represent them using the set of graphical symbols presented in figure 1.1. These symbols are used to denote the various conceptual elements involved in the execution of an MAS based on an MAOP approach. In addition to these graphical representations, we use UML-like diagrams to represent the concepts that participate in the definition of the dimensions structuring the MAOP approach. Various notations from UML are not used in these figures, in order to keep them simple and easy to read. For instance, we decided not to display cardinalities on the composition links connecting the different concepts.

The book is accompanied by a website (http://jacamo.sourceforge.net/book) from which all the examples and complete systems can be downloaded and run. It should be stressed that the platform for running these systems is freely available and is open source. It is available at http://jacamo.sourceforge.net. The platform is based on programming tools that have been shown to be robust over many years of research and development, and are used by researchers and practitioners around the world.

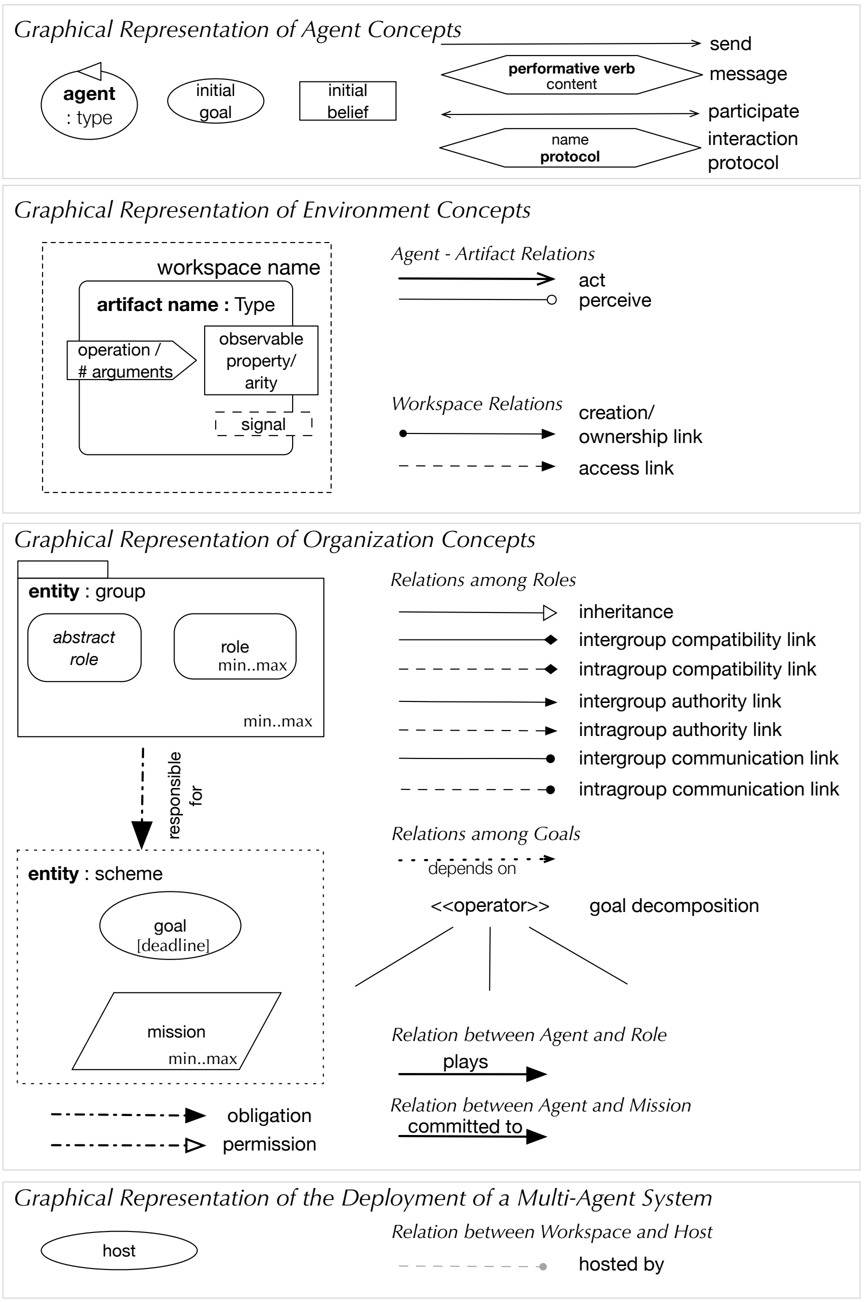

Graphical Representation of Agent Concepts
agent
: type
initial goal
initial belief
send
performative verb
content
message
participate
name
protocol
interaction protocol
Graphical Representation of Environment Concepts
workspace name
artifact name : Type
operation / # arguments
observable property/ arity
signal
Agent - Artifact Relations
act
perceive
Workspace Relations
creation/ ownership link
access link
Graphical Representation of Organization Concepts
entity : group
abstract role
role
min..max
min..max
responsible for
entity : scheme
goal
[deadline]
mission
min..max
obligation
permission
Relations among Roles
inheritance
intergroup compatibility link
intragroup compatibility link
intergroup authority link
intragroup authority link
intergroup communication link
intragroup communication link
Relations among Goals
depends on
<<operator>>
goal decomposition
Relation between Agent and Role
plays
Relation between Agent and Mission
committed to
Graphical Representation of the Deployment of a Multi-Agent System
host
Relation between Workspace and Host
hosted by

Figure 1.1
Graphical symbols used in the book.

2 An Overview of Multi-Agent Oriented Programming

In this chapter we introduce the main concepts of multi-agent systems (MAS), and we provide an overview of multi-agent oriented programming as a comprehensive approach to programming MAS.

2.1 Multi-Agent Systems

Multi-agent systems are a paradigm for modeling and engineering complex systems. By *paradigm* we mean a set of concepts, techniques, technologies, and methodologies. Modeling means creating formalized representations (models) based on these concepts that capture essential aspects concerning the structure and behavior of the target systems. Modeling is an important aspect of two main contexts in which MAS are used: (1) simulations, in which models are useful to describe and simulate existing complex systems, either natural or artificial, to analyze their properties; and (2) engineering, which is more oriented to the design and development of systems and applications. An example for the simulation case is an agent-based model for traffic simulations, in which cars are modeled as agents and the city and street signs are part of the agent environment. An example for the engineering case in the same domain is a model of a smart city in which autonomous unmanned cars are designed as agents, interacting with other agents and digital services representing the smart infrastructure. The engineering perspective is the one adopted in this book.

Using MAS, a complex system is modeled (and designed, and programmed) as an organization of autonomous agents situated and interacting in some (logical) environment. *Agents* represent the decision-making entities of the systems; that is, agents are entities designed to autonomously pursue some goal, encapsulating for that purpose a logical control flow, and making decisions about how to behave and interact. In the smart city example, the goal of the agent representing the autonomous unmanned car could be about reaching some target place, choosing the most convenient path according to the preference of the human users. Agents are

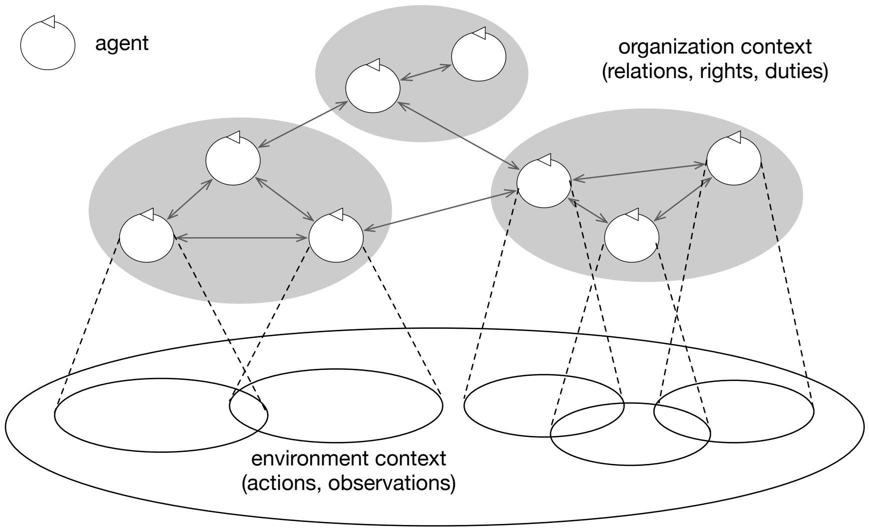


Figure 2.1
A representation of multi-agent systems inspired by Jennings (2001).

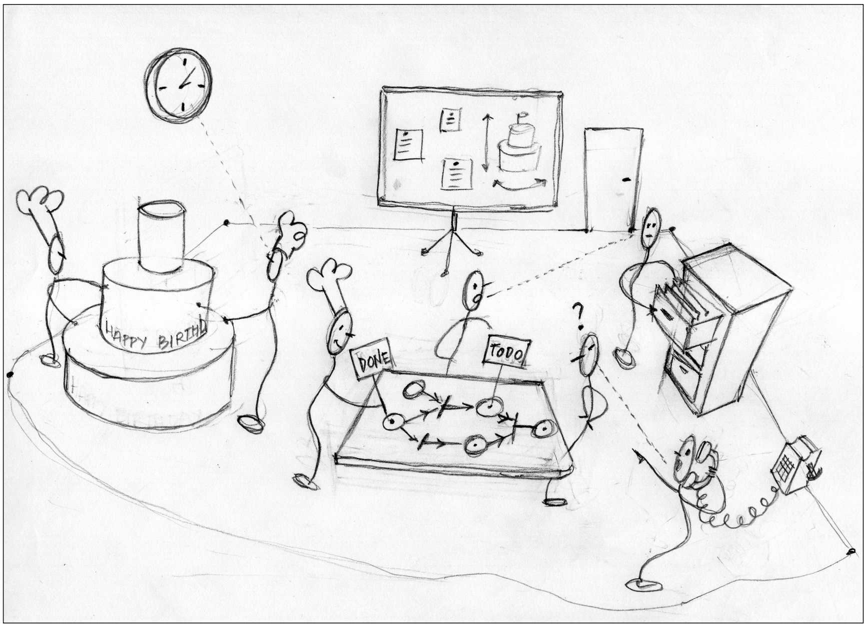


Figure 2.2
Analogy between MAS and human system (a bakery workshop scenario).

logically situated in an *environment*, which they perceive and act upon in order to achieve their goals. The environment represents the *context* in which an agent acts. In the smart city example, the environment could be both the physical environment such as the street and also the digital services such as shared situated message boards communicating relevant information.

The environment can be large-scale and distributed; nevertheless, an agent can observe and act upon only a portion of it at any given time (see figure 2.1). The analogy with the real world is quite straightforward: people (i.e., agents) perceive and act in some environment, decide what to do in order to achieve their goals, interact with other people, and so forth. As in the human case, the environment can be seen also as the set of resources and tools that agents can share and use to do their tasks. Figure 2.2 shows a bakery as an example, which is also used in subsequent chapters.

A multi-agent system involves multiple agents communicating by means of some high-level agent communication language (ACL) and cooperate to achieve shared goals. The organization (figure 2.1) explicitly captures the main aspects that characterize the tasks (and functionalities, behavior, and properties) of the system as a whole. In the smart city example, for instance, the set of agents representing the autonomous unmanned buses could form a group with both individual missions and shared goals and with the rights and duties that constrain their behavior and interaction within the organization and between organizations.

MAS Paradigm vs. Existing Paradigms

Agent-oriented modeling shares some characteristics with *object-oriented* modeling. Both adhere to the principle of information hiding and recognize the importance of interactions. Like objects, agents hide internal state and implementation details, and message passing based on some shared agent communication language is the primary means for agents to communicate each other.

A first key difference is that agents encapsulate a logical thread of control that make them *active* (rather than passive like objects), embedding a logical thread of control. Different from objects, agents encapsulate state, behavior, and *the control of that behavior*. This also impacts the communication/interaction model, which is strictly asynchronous, so that an agent that sends a message to another agent does not transfer its control flow (as in the case of method call between passive objects), and the control remains encapsulated. Analogously, an agent processes messages with its own thread of control.

From this point of view, agents are similar to *actors*. However, actors are strictly reactive entities; they act only in reaction to receiving a message. After executing the handler (or method) associated with a message, if there are no other messages to process, the actor becomes idle. Conversely, agents are goal-oriented proactive entities, so they act in order to perform their tasks even if no messages are received.

Another main difference between agent-oriented modeling and object/actor oriented modeling is the environment as first-class abstraction. In a pure object-oriented programming world, everything is modeled as an object. In a pure actor world, everything is modeled as an actor. In agent-oriented modeling, agents communicate with other agents but interact also with an environment, by means of actions and perception. Therefore, agent-oriented modeling does not involve modeling everything as an agent. There is a separation of concerns between those parts of a system representing/encapsulating decision making and those parts of the system representing entities to be controlled, providing actions and observable state/events for that purpose.

2.2 Multi-Agent Oriented Programming

In principle, any programming technology could be used to implement an MAS, given a clear description of the model. However, the risk is to have an agent-centred interpretation of figure 2.1 in which the environment and/or the organization contexts are represented and managed in the mind of the agents. The adoption of programming languages that directly provide first-class programming abstractions greatly simplify our task, making it possible *to keep the level of abstraction coherent from design time to development time and also at runtime*.

multi-agent oriented programming

Multi-agent oriented programming is an approach to programming MAS that promotes the use of first-class programming abstractions that concern three main dimensions that characterize a multi-agent system, namely, the *agent* dimension, the *environment* dimension, and the *organization* dimension (shown in figure 2.3). Every dimension defines a set of concepts and first-class abstractions capturing different concerns of the MAS.

agent dimension

The *agent dimension* groups concepts and programming abstractions for the definition and programming of the agents participating in the system. The agent concept is the key abstraction, used to program the decision-making entities that are able to provide local flexibility and dynamism by reacting to events while proactively directing their behavior to reach future states of the system so as to satisfy

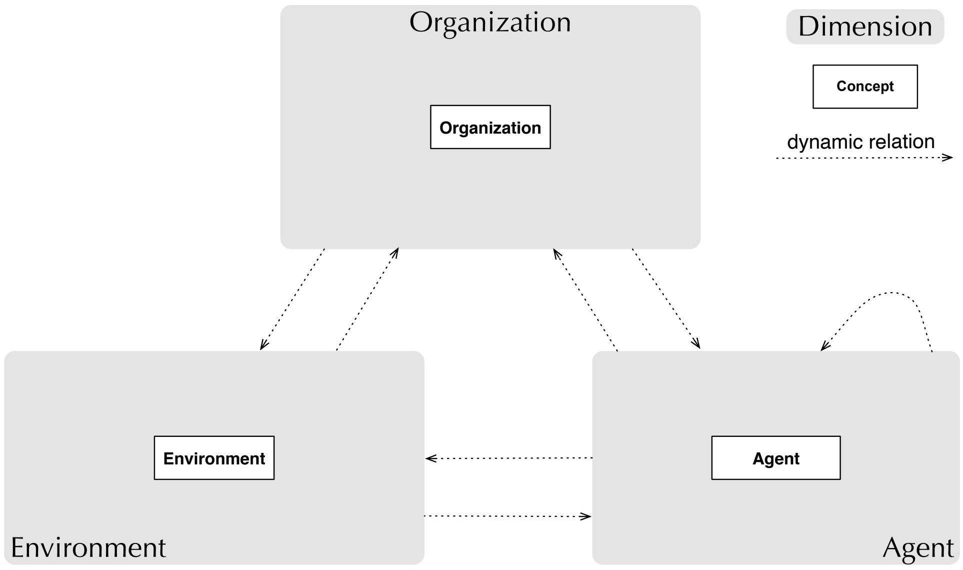


Figure 2.3
Dimensions of Multi-Agent Oriented Programming.

certain *goals*. The resulting software entity has its own logical thread of control to autonomously achieve those goals, interacting with the environment, other agents, and the organization that regulates the overall system.

autonomy

One of the most defining features of agents in this view is *autonomy*. Autonomy is inseparable from the agent concept in the sense that an agent is designed to reason about *what* to achieve and, importantly, *how* to do so given the current system circumstances. To this purpose, an agent features *proactivity*, that is, the capability of taking the initiative about the actions to perform in order to achieve its goals; *reactivity*, that is, the capability of promptly adapting its behavior depending on events perceived from the environment; and *social ability*, that is, the capability of communicating and cooperating with other agents. At the organization level, autonomy implies decentralization of control, so that an agent (and its behavior) is typically *influenced* but not *regimented* by either other agents or organizational norms. Norms are used to define the expected behavior in a social system, but autonomous agents may choose not to abide.

proactivity

reactivity

social ability

environment dimension

The *environment dimension* offers concepts and first-class abstractions for the definition and programming of the distributed resources and connections to the real world shared among the agents. If agents are useful to model autonomous goal-oriented entities, the environment as first-class abstraction is useful to model any elements that can be used or controlled by agents to achieve their goals. The environment abstraction is what makes agents *situated*, that is, logically placed in a context that provides them a set of actions to affect the environment and exposes some kind of observable state and events that agents can perceive.

situated

organization dimension

The *organization dimension* collects all the necessary concepts taking part in the definition and programming of relations, joint tasks, and policies among agents interacting in a shared environment. The central concept of *organization* defines the structuring, coordination, and regulation of the agents working together. Mirroring the autonomy feature of the agent dimension facing the dynamicity of the environment dimension, the most defining features of the organization in this view are those of *coordination* and *regulation* that help face the *openness* requirement. Coordination refers to the support for the work of multiple agents that depend on each other to achieve whatever they individually or collectively aim to achieve. Regulation refers to the support for taming the autonomy of the participating agents. The regulatory aspect is typically carried out with the use of *norms* (social rules that agents are expected to follow and might be somehow punished for not following). Programming the organization opens the possibility for agents to reason on their relations, joint tasks, and policies, and to decide on their adaptation as well as on the compliance to the resulting constraints.

coordination
regulation

2.3 Main Abstractions

Each MAOP dimension includes a variety of concepts and programming abstractions that support the development of MAS. Here we provide a bird's-eye view intended to serve as a map (see figure 2.4), first identifying the main abstractions for each dimension and then remarking on the connections gluing the dimensions. In subsequent chapters we look at the dimensions in more detail.

goal

Agent dimension In the agent dimension, the concept of *goal* is very important; it provides the means for the representation of some state of affairs that the agent would like to bring about. An explicit representation of the long-term goals that agents have to achieve is essential for autonomy and proactivity. In some cases, it corresponds to making explicit the design goals of each agent. Similarly important for autonomy is that agents are able to make rational choices on which goals to pursue at runtime as well as the choice of the means to use in order to achieve those goals.

beliefs

Those choices and ensuing behavior are guided by agent *beliefs*. Beliefs are simply explicit representations of information available to the agent, on which the agent can reason; the term is used to emphasize that information available to an agent might be incorrect and/or incomplete in typical multi-agent systems. Beliefs are acquired by perceiving the state of the environment and by communicating with

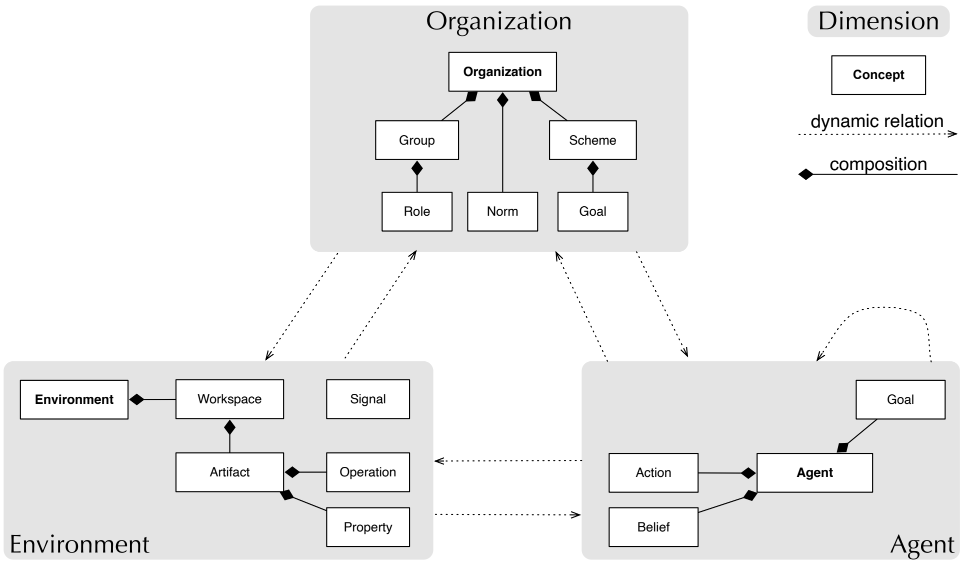


Figure 2.4
Dimensions and main concepts for MAOP.

other agents, for example. The main point of an agent is the *actions* it will take in order either to change the state of the environment so as to achieve its goals or to interact with other agents or elements of the environment. *actions*

Environment dimension In the environment dimension, the concept of *workspace* is used to define topological or symbolic regions of the environment that are populated by a set of *artifacts* and by agents. An *artifact* represents a real or conceptual environment resource through a set of *operations*, which agents can use to execute actions, and *properties*, which agents can observe to acquire beliefs. Recall that environment entities are not autonomous nor proactive like agents. Using these operations on the artifact instances currently available in the system, agents can change the state of the environment as well as they can observe it, given the properties provided by those artifact instances. Artifacts are useful to modularize an environment, making it dynamic—artifacts can be created and destroyed dynamically (by agents). By means of artifacts, the environment provides a layer of software abstractions to support agent interaction through shared resources in an implicitly controlled way. *workspace* *artifacts* *operations* *properties*

Organization dimension In the organization dimension, the concept of *group* is used to provide social structure to the system, giving support for the definition of *group*

roles
norms
organizational goals
social plans
social schemes

the expected coordinated behavior in the system as well as the rights and duties that have to be fulfilled by the agents. In a group, *roles* determine the interactions and relations taking place within the group. They also take part in the definition of the rights and duties expressed as *norms*. The expected behavior for rights and duties is expressed as sets of *organizational goals* used in the definition of *social plans*, which are goal decomposition trees contained in *social schemes* that are executed under the responsibility of groups. A social plan makes explicit the expected coordinated achievement of the goals when rights and duties expressed by norms are fulfilled. When organizations are imposed on a society of agents, the agents that play roles in the assigned groups are required to work together, coordinating their actions in order to achieve an organizational goal of the system by abiding by the rights and duties imposed by the norms corresponding to their roles.

As a key aspect of the MAOP approach, all these abstractions are *kept at runtime* by the platform running the MAS (which is JaCaMo in our case). Thus a software system programmed using MAOP can dynamically change and reorganize in different ways, for instance, by means of the runtime creation of organization and artifact instances in the environment, or rather agents joining and leaving organizations through the roles they choose to adopt. This is useful for many of the features we need in current modern systems, as we discuss subsequently in this chapter.

2.4 Integrated View

multi-agent system
Interaction

Putting the dimensions together to program a system thus results in a *multi-agent system*, which is a set of agents, one environment, and a set of interacting organizations, as shown in figure 2.5. *Interaction* is an essential aspect of the multi-agent oriented programming approach. Multi-agent systems are all about interaction among the autonomous agents that communicate with each other and interaction between them and the environment, through which they also interact with the organizations that regulate and coordinate part of their activities. These interactions occur when the system is executing and are thus represented in figure 2.5 as *dynamic* relations between the dimensions.

Dynamic relations (figure 2.5) connect the agent, environment, and organization dimensions, generating a closed and rich cycle of interacting instances of concepts belonging to the different dimensions.

communicate

Agents may choose to *communicate* with other agents, creating a dynamic relation between them, making them able to interact with other agents, especially other agents that currently take part in the same organizations, but not exclusively. The way this direct agent-to-agent interaction takes place is based on speech act theory, so communication is seen as an action that changes the mental state (e.g., the beliefs and goals) of both sender and receiver agents. This interaction is thus rather special

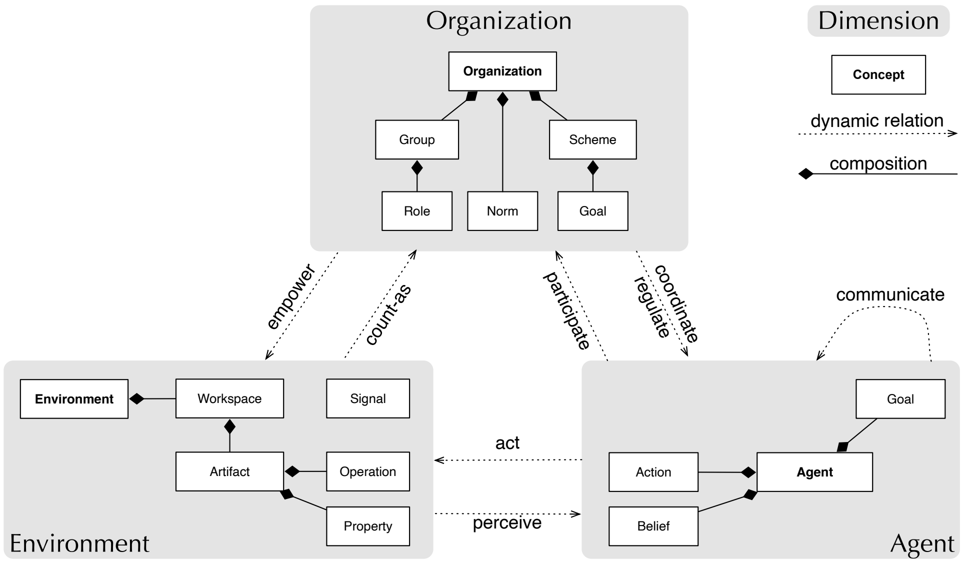


Figure 2.5
Dynamic relations among instances of concepts of the MAOP dimensions.

in comparison with other approaches to communication in distributed computing. This is discussed in more detail in chapter 4.

Interaction in a multi-agent system is not limited to agent-to-agent communication; it also happens between agents and the environment. As mentioned previously, agents can *perceive* (i.e., *observe* or *sense*) the artifacts situated in the environment and react to that perception. Agents can also *act* upon the artifacts to change their state. Within a workspace, agents and artifacts are visible to each other, although agents may ignore or focus on particular artifacts at will to help scalability. Agents interact with the environment as well as with other agents. By means of the environment, agents can interact among themselves *indirectly*. For instance, an agent performs the action *open door* in a kind of door artifact that is observed by another agent that perceives and represents it as a belief on the environment and may trigger a goal leading the agent to enter the room.

perceive
act
indirectly

Changes in the state of the environment may also *count as* changes in the state of the organization. In order to support the joint work of agents, an organization needs to be attentive to what agents are doing in the environment. For example, if there is a dependency between the tasks of two agents, when the first task is perceived as having achieved some goal through the state of the environment, the organization can require the agent responsible for the second task to engage in some course

count as

of action in order to perform the second task as it previously committed. This is one of the ways that an organization can regulate and coordinate agent activities. empower Conversely, organizations instantiating the organization dimension may *empower* the elements of the environment by allowing them to control and regulate actions or perception of the agents. This dynamic relation is a practical way of situating organizations in an environment, as happens for the agents, regulating some part of the environment (e.g., a traffic light at a crossroads) in a particular way and ruling it differently in other parts.

Finally, the dynamic relations between the agent and organization dimensions refer to the influence that the organizations may have on the agents: they can *regulate* and *coordinate* agents for dynamically created tasks. The coordinate relation refers to the management of dependencies between activities carried out by agents. The regulate relation, in turn, refers to exerting control on those activities. Of course, as agents are autonomous, this can happen only if they actively *choose to participate* in one or more of the currently existing organizations in the system.

regulate

coordinate

2.5 Overcoming Challenges

We conclude this chapter by going back to the challenges introduced in section 1.2 in the preceding chapter and providing an overview of how they are handled by the multi-agent oriented programming approach described in this book.

Autonomy MAOP proposes a structured approach to *autonomy* with a clear separation of concerns. First of all, agents are the place where autonomy is considered, whereas the artifacts, which are active or passive entities situated in the environment, are considered nonautonomous. The concepts of the agent dimension support autonomy in allowing designers to concentrate on explicitly represented goals and agents to reason about the most appropriate courses of action to achieve them. Furthermore, various AI techniques can be directly plugged into the agent architecture so as to increase autonomy. This separation of concerns between agents, autonomous entities, and artifacts, nonautonomous entities, takes all its importance in defining the organization, which uses abstractions for the coordination and regulation patterns that target the autonomous entities. It allows multiple autonomous agents to coordinate their action, thus in some sense taming their autonomy so that the system works coherently. Thus MAOP supports the programming of autonomous systems with the agent dimension addressing the definition of autonomous entities themselves, the environment dimension targeting the definition of the shared entities that are the sources of perception and targets of actions of the autonomous entities, and finally the organization dimension expressing the required control of the autonomous entities in the shared environment.

Decentralization and distribution Interaction is a central mechanism in MAOP. It fosters loose coupling between the autonomous entities and thus ensures the possibility of developing and deploying *decentralized systems*. On one hand, the use of sophisticated direct interactions between agents through speech acts enables autonomous agents to act on other autonomous agents. On the other hand, the use of indirect interactions allows independence of agents using the environment as the shared medium of interaction between them. Using organizations, which allows for the declarative programming of coordination and regulation patterns, it is possible to control and structure such decentralized systems. Even if these patterns are shared among the agents, no central control point is introduced. The decision-making process can be kept decentralized. Organizations address new challenges in software development: to be able to deal with coordination of the work of vast numbers of autonomous agents (both software and humans) and to maintain the decentralization of such coordination.

Complementary to decentralization, the MAOP approach provides many opportunities for *distribution*. Thanks to the modularity promoted by the separation of concerns, the modular components that can be considered in the MAOP perspective are, besides agents, workspaces, and artifacts in the environment dimension and organizations, groups, and schemes in the organization dimension. Thus distribution may concern running on different machines the agents, workspaces, artifacts, organizations, groups, schemes, and other components. Multiple organizations with multiple running agents and various parts of the environment might be distributed across different computing platforms, which also helps support scalability, arguably the "holy grail" of modern computing.

Openness Addressing the *openness* challenge through an MAOP approach helps identify and structure the appropriate answers to the various evolutions of the agent (e.g., entry/exit of agents), environment (e.g., creation/deletion of artifacts and the topology of workspaces), and organization (e.g., changes in the patterns of coordination or regulation in the organization and creation/deletion of organizations) dimensions. In MAOP, agents, artifacts, workspaces, and organizations with their groups and schemes are meant to be created, discovered, and possibly disposed of by the agents themselves at runtime. This is a basic way in which MAOP supports dynamic extensibility (besides modularity) of the agents, environment, and organizations. For instance, a new agent can be able to fit in the global functioning resulting from the various interactions of the system in terms of agents, artifacts, and organizations. Agents can cope with incomplete knowledge and control and can interact with other agents, with artifacts of the environment, or with norms of the organization that were not known at design or deployment time. The system can adapt its global functioning to a malfunctioning or shutting down of an agent. It can be prepared to deal with any number of agents. The environment and its artifacts can evolve and integrate incrementally new functionalities (e.g., storage and processing resources) through deployment of new artifacts. The organizations can evolve and integrate new norms, new social plans, and new structures according to the actions undertaken by the agents.

Facing continuous evolution raised by entities that may be unknown at the system initialization time, an MAOP approach offers various means to support and ensure coherent behavior of the system. Explicit and declarative representations of organizations help to regulate and control such open systems, defining behavior boundaries as *admissible behavior* corresponding to the coordination and regulation strategies used in the system. Besides this soft control, environment abstractions limit the repertoire of actions that can be undertaken in the physical environment via the artifacts. They help define the behavior boundaries as *possible behavior* corresponding to the actions offered by the shared storage and processing artifacts as well as the mechanisms to monitor and discover entities of the system. It is thus possible to ensure that the behavior produced by agents, even if developed by stakeholders other than the multi-agent system's stakeholders, belong to the set of possible and admissible behavior. From another point of view, it also supports the changing and adaptation of the coordination and regulation strategies or of the storage and processing units without changing the agents.

Heterogeneity Each of the multi-agent oriented modeling dimensions introduces means for defining various modules that introduce *heterogeneous* representations and dynamics at the system level that could potentially have been programmed by different people or that indeed represent the interests of different, possibly competing, companies.

In general, agents in a system have different capabilities, possibly different architectures, and other differences. Artifacts encapsulate different resources, which may be physical or digital. Organizations express various different coordination and regulation models. Therefore, typically, an MAS along an MAOP approach is highly heterogeneous.

As previously mentioned, to face heterogeneity, interoperability is a sought-after property. According to IEEE, "interoperability refers to the ability of two or more systems or components to exchange information and to use the information that has been exchanged" (Geraci et al. 1991). Several variations of the general idea of interoperability have been proposed (e.g., Morris et al. 2004; Tolk and Muguira 2003) to extend it to a broader set of concerns: considering technical and business interoperability levels, any interaction among entities (be they human, software, things, or mixed populations) requires a common, or shared, notion of the *environment* and of the concepts used in the *organization* and *regulation* of that environment.

As given in chapter 1, section 1.2, heterogeneity often relates to concerns about interoperability of different systems. In MAOP, interoperability concerns are identified and split according to the interfaces between each of the MAOP dimensions. These include the following examples:

- *Agent-Agent*, which defines common communication languages for direct interaction between heterogeneous agents;
- *Agent-Organization*, which defines organization representations to enable agents to read, act, and reason about the coordination and regulation patterns governing their behavior; and
- *Agent-Environment*, which defines explicit usage manuals with the action and perception repertoire available in the artifacts of the environment to enable agents to read, use, and reason about the available actions.

Adaptability For the challenge of *adaptability*, the MAOP approach allows a developer to identify and address the problem at the different time and duration scales taking place in a multi-agent system: short-term repetitive activity consisting of action and perception connecting agents to their environment, long-term activity corresponding to management of goals in the agents to decide on their actions, and medium-term activity represented in the organization defining strategies and policies that control and regulate the management of goals and decisions in the agents and thus their actions in the shared environment.

The agent programming model adopted, which is based on the Belief-Desire-Intention (BDI) architecture, allows choosing and adopting different plans to achieve the same goal *depending on the context*, which makes it straightforward to develop a context-aware behavior. The execution of a plan can be interrupted dynamically, for example, in the case of failure or changes in the environment that call for different strategies, or even different goals to be adopted.

The use of first-class abstractions to represent, modularize, and manipulate the environment makes it possible for agents to decide themselves to adapt the set of resources and tools to be used to achieve their goals.

The combination of environment and organization dimensions brings context-awareness and adaptability at the organization level. As for agents, the state of the real-world environment can be considered to count as organizational facts that can trigger or hinder regulation or coordination patterns. Representing the particular set of agents that are currently taking part in the available groups (i.e., the agents that actively decided to adopt roles in particular groups of the organization) helps dealing with context awareness of the global coordination and regulation patterns governing the system, organizations in particular. Furthermore, organizations have explicit representations for what is going on in their interrelated structures in which agents participate, the current state of the organization, so that agents can act accordingly.

Explainability Even for *explainability*, the MAOP approach allows dealing with it at different levels, from micro-level to macro/global. For instance, thanks to explicit goal representation, agent-based systems are developed so that particular choices on courses of actions made by the individual agent can be explained to users on demand, which is of great importance when we deal with autonomous systems. In the same way, organizations provide explicit representation of regulation and coordination schemes as well as authority and communication structures that could be the basis for explaining the global structure of collective autonomous systems as well as their global functioning.

AI integration As pointed out in section 1.2, *AI integration* is an important feature to develop autonomous systems in unpredictable and evolving environments. In this case, MAOP provides a disciplined approach to embed and exploit AI techniques. In particular, the agent programming model adopted in this book is based on the BDI model/architecture, and research contributions in the literature describe extensions to integrate techniques such as planning and reinforcement learning. Nevertheless, AI can be introduced also as a service, wrapped into environment resources (artifacts) that agents can exploit to handle specific tasks (e.g., speech recognition). Besides the individual level, planning and learning can be thought about at the MAS level, for instance, implementing multi-agent planning and multi-agent reinforcement learning strategies described in the literature. Chapter 11 provides an overview of these research strands.

2.6 Wrap-Up

In this chapter we first introduced multi-agent systems as the reference paradigm used in this book to model and design software systems, and then we provided an overview of the key ideas about multi-agent oriented programming, a multi-dimension programming approach for developing MAS. We went back then to some main challenges that were discussed in chapter 1 about the kind of applications and systems for which MAS are suitable, discussing in general how MAOP key ideas are effective in tackling such complexities. In the remainder of the book we delve into MAOP in theory and practice, starting from the agent dimension (chapter 4). Before that, in the next chapter we get started with JaCaMo, the platform that is used to write and run the examples and programs appearing in this book.

2.7 Bibliographical Notes

The research community working on agents and multi-agent systems is quite broad and related to different existing communities from (distributed) artificial intelligence, cognitive science, robotics, software engineering, simulations, and others. The main venue for research in the area is the International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS), which was set up in 2002 as a joint effort of the Autonomous Agents conference which was focused on the individual agent level; the International Conference on Multi-Agent Systems (ICMAS) which was focused on multi-agent aspects; and the workshop series on Agent Theory, Architectures, and Languages (ATAL) which was focused on the theory and practice of programming and developing agent-based systems. Some workshops organized in the context of AAMAS have been explicitly focused on programming and engineering MAS, in particular on Programming Agents and Multi-Agent Systems (ProMAS); on Agent-Oriented Software Engineering (AOSE); and on Declarative Agent Languages and Technologies (DALT). In 2012, these three workshops joined efforts in a single event called Engineering MAS (EMAS). Proceedings of these workshops are available as *Lecture Notes in Artificial Intelligence* (LNAI), published by Springer.

Reference books that provide a broad introduction to this field include Weiss (1999), Wooldridge (2009), and Ferber (1999). Another reference book providing a more game-theoretical perspective is Shoham and Leyton-Brown (2008). Some books focus more on engineering and programming aspects, like this one. Agent-oriented modeling and design is treated in Sterling and Taveter (2009) and Padgham and Winikoff (2004), whereas Bordini et al. (2009) provide a broad collection of approaches to multi-agent programming. Books about specific agent-oriented programming platforms and technologies include Bordini et al. (2007) about programming BDI agents using Jason and Bellifemine et al. (2007) about developing multi-agent systems using JADE.

Finally, some research papers in the literature provide a broad picture about MAS and MAOP. We consider the following short selection related to the topics of this book. Jennings (2001) provides an introduction to the agent paradigm for building complex software systems. Wooldridge and Jennings (1995) provide an overview of the main points concerning the theory and practice of intelligent agents. Iglesias et al. (1999) and Bordini et al. (2006) provide surveys about agent-oriented methodologies and programming languages and platforms for MAS, respectively — although they are not recent, they are still valuable references from a historical perspective. The idea of adopting a multidimensional approach in modeling, programming, and engineering MAS was first proposed in the Vowels decomposition paradigm (Demazeau 1995). Boissier et al. (2013, 2019) provide an overview on multi-agent oriented programming and the integration of multiple programming dimensions.

Further references and suggestions about aspects of agent and MAS programming are given in subsequent chapters discussing the specific topics.

3 Getting Started

In this chapter we introduce JaCaMo, the particular platform adopted in this book for hands-on multi-agent oriented programming. This platform supports practical programming based on the abstractions introduced in the previous chapter: programming organized agents situated in a shared environment. JaCaMo is built on top of three existing platforms that have been developed for years (Boissier et al. 2013, 2019), namely Jason (Bordini et al. 2007) for programming agents, CArtAgO (Ricci et al. 2009) for programming environments, and Moise (Hübner et al. 2007) for programming organizations.

The accompanying site for this book, available at

http://jacamo.sourceforge.net/book

has the source files from which all examples in this book were taken. It also has pointers to further instructions on how to run those systems. The reader should follow this chapter and run the code at the same time, to become familiar with the use of the platform, which will be useful throughout the book. For this hands-on tutorial, however, rather than download the example files, the reader may prefer to work interactively by simply following the instructions on creating new applications.

The classic Hello-World example has a multi-agent version in this chapter. We start with the simplest application we can write in JaCaMo and improve it to progressively show some of the most important aspects of the programming language and the platform.

Instructions to create, edit, and run JaCaMo applications are found at

http://jacamo.sourceforge.net/doc/install.html

3.1 Single-Agent Hello-World

We start with a system that has a single and very simple agent that just prints out a message using the following plan written in Jason (and stored in a file called `hwa.asl`):

```
+!say(M) <- .print(M).
```

plan This *plan* can be read as "whenever I have the goal `!say(M)`, achieve it by printing the value of variable `M`" (`M` is a variable because it starts with an upper-case letter).

To run the agent, JaCaMo uses application files (of which the names end with `.jcm`). In our example the application file is `sag_hw.jcm`, in which we give a name goal to agent (`bob`) and an initial *goal* (`say("Hello World")`). The content of this file, represented graphically in figure 3.1, is as follows:

```
mas sag_hw {                    // the MAS is identified by sag_hw
  agent bob: hwa.asl {  // initial plans for bob are in hwa.asl
    goals: say("Hello World")           // initial goal for bob
  }
}
```

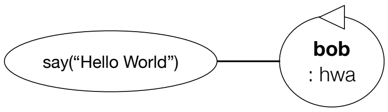


Figure 3.1
Single-agent Hello-World configuration.

The result of the execution is

```
JaCaMo Http Server running on http://192.168.0.15:3272
[bob] Hello World
```

To better understand the output, the steps in executing the application file (.jcm) are as follows:

1. An agent named `bob` is created with initial beliefs, goals, and plans as determined from the contents of the file named `hwa.asl`.
2. The goal `say("Hello World")` is delegated to bob, creating the event `+!say("Hello World")`.
3. The plan in file `hwa.asl`, as shown previously, is triggered and used to handle this event.
4. The execution of the plan produces the output `[bob] Hello World` as the result of performing the internal action `.print`.

5. The agent continues to run but has nothing to do, because it is the only agent in the system and has not generated any further goals itself or changes in the environment that might lead it to further action.
6. As shown in the execution output, there is a URL to inspect the current state of the agents (which includes their beliefs, intentions, and plans), and we subsequently see that the same applies to environment and organizations.

3.2 Multi-Agent Hello-World

We now have two agents, bob and alice. Agent bob prints "Hello" and alice prints "World." In order to create both agents from the same code (as in the previous example, having only one plan), we can use the following application file:

```
mas mag_hw {
    agent bob: hwa.asl {
      goals: say("Hello")
    }
    agent alice: hwa.asl {
      goals: say("World")
    }
}
```

However, the result of the execution could be as follows:

```
[alice] World
[bob] Hello
```

Agents run concurrently and asynchronously pursue their goals, and thus this initial implementation cannot guarantee the order of the printed messages. Some coordination is required so that bob prints first and alice next. We can solve the problem with bob sending a *message* to alice as soon as its message is printed (see figure 3.2). The new program is as follows (to be included in a file named `bob.asl`): *message*

```
+!say(M) <- .print(M);
            .send(alice,achieve,say("World")).
```

By sending an *achieve* message to alice, bob delegates the goal `say("World")` to alice. She uses the plan `+!say(M) <- .print(M).` to achieve the goal, as previously.

Because alice's goal now comes from bob rather than the system initialization, the application file has to be changed as follows:

```
mas mag_hw {
    agent bob { // file bob.asl is used
      goals: say("Hello")
    }
    agent alice: hwa.asl
}
```

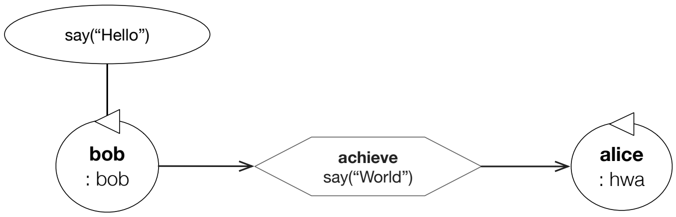


Figure 3.2
Coordination by communication.

3.3 Hello-World Environment

artifact The example now considers an environment with a `board` *artifact*, as a blackboard that agents can use to write messages and to perceive messages written on it. In this version of the Hello-World example, bob writes a message "Hello" on the `board` blackboard; and alice, who is observing the blackboard, writes the message "World" as soon as it acquires the belief that the message "Hello" has been written.

workspace Environments are structured into workspaces; all agents within a *workspace* have shared access to all artifact instances in that workspace. In the application file, we can specify the initial set of artifacts and workspaces to be created when the MAS is spawned. In this case, the `sit_hw.jcm` file is as follows:

```
mas sit_hw {
  agent bob {
    join: room              // bob joins workspace toolbox
    goals: say("Hello")
  }
  agent alice {
    join: room        // alice also joins workspace toolbox
    focus: room.board       // and focus on artifact board
  }
  workspace room {         // creates the workspace toolbox
    artifact board: tools.Blackboard // with artifact board
  }
}
```

The initial configuration includes a workspace called `room`, hosting a `board` artifact of type `tools.Blackboard` (see figure 3.3). Both agents *join* the workspace `room` at initialization in order to access the `board` artifact. Agent alice *focuses* on

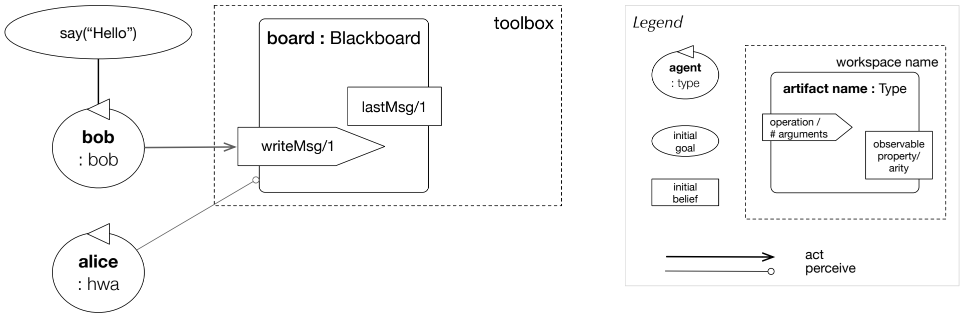


Figure 3.3
Coordination using the environment.

(i.e., observes) the artifact.[1] The focus is needed so that agent alice is attentive to changes in the observable properties of that artifact: when something is written on the board, the next time alice senses the environment a *belief* corresponding to the observed artifact property will be automatically created, and she can then react to it.

belief

Artifacts are implemented in Java. The source code (in file `Blackboard.java`) of the simple blackboard artifact is as follows:

```
package tools;
import cartago.*;

public class Blackboard extends Artifact {
  void init() {
    defineObsProperty("lastMsg","");
  }
  @OPERATION void writeMsg(String msg) {
    System.out.println("[BLACKBOARD] " + msg);
    getObsProperty("lastMsg").updateValue(msg);
  }
}
```

Java classes are used as templates for defining artifacts, using annotated methods to define artifact operations and predefined methods inherited by the `Artifact` API to work with observable properties and other artifact mechanisms.

1. In fact, agents can dynamically decide which workspaces to join and which artifacts to focus on; in that case, the actions to join a workspace and to focus on an artifact appear in the source code of the agent. This is detailed in chapter 5.

The source code for bob in this case becomes

```
+!say(M) <- writeMsg(M).

{ include("$jacamoJar/templates/common-cartago.asl") }
```

That is, the agent uses the action `writeMsg` provided by the artifact to write the message on the blackboard. The `include` instruction loads some useful plans into bob's plan library. The source code for alice is

```
+lastMsg("Hello") <- writeMsg("World!").

{ include("$jacamoJar/templates/common-cartago.asl") }
```

The agent (who is observing the `board`) writes the message "World" as soon as it has the belief that the last message written on the blackboard (made observable by means of the `lastMsg` *observable property*) is "Hello."

observable property

```
[alice] joined workspace room
[alice] focusing on artifact board (at workspace room)
using namespace default
[bob] joined workspace room
[BLACKBOARD] Hello
[BLACKBOARD] World!
```

The execution of the application file produces a result similar to the previous ones, except that now it is the blackboard artifact printing out the messages and no communication between bob and alice is required.

3.4 Hello-World Organization

organization

We now organize the set of agents to produce the "Hello World" message. As presented in the preceding chapter, the organization can be used to regulate and coordinate the agents. Although the example is simple, the use of an *organization* facilitates the changing of a specified coordination and regulation pattern. In our example, a coordination pattern is used to achieve the goal `show_message`, which should be achieved by the two agents working together and thus is a collective goal. To distinguish such a goal from an agent goal, we call it an *organizational goal*.

organizational goal

scheme

missions

We use a social *scheme* to program how the `show_message` organizational goal is decomposed into subgoals that are assigned to the agents (as shown in figure 3.4). For the decomposition, the `show_message` goal has one subgoal for each word of the message. For their assignment to agents, we create *missions*, in this case one for each subgoal. In order to participate in the scheme execution, agents should *commit* to a mission and achieve the corresponding goal(s) of that mission. Committing to a mission is a form of promise to the group of agents collectively working on a scheme: "I promise that, when required, I will do my part of the task." When agents

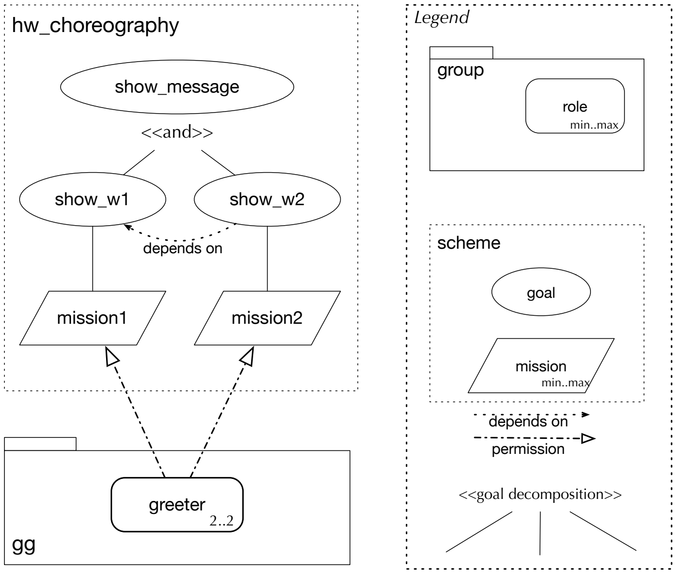


Figure 3.4
Coordination by organization: Hello-World organization specification.

have committed to all missions, the scheme can be performed with the guarantee that, at least in principle, we have enough agents to work on all required subgoals.

This organization example also defines a single *role* that all agents will play: the role `greeter` played in a *group* type identified by `gg` (for "greeting group"). Agents playing this role (and only they) are permitted to commit to the missions of the scheme. *role* *group*

The implementation of this organization is written in XML as follows:

```
<?xml version="1.0" encoding="UTF-8"?>

<organisational-specification
  id="hello_world"
  os-version="0.8"

  xmlns='http://moise.sourceforge.net/os'
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xsi:schemaLocation='http://moise.sourceforge.net/os
                      http://moise.sourceforge.net/xml/os.xsd' >

<structural-specification>
  <group-specification id="gg">
    <roles>
      <role id="greeter" max="2"/>
    </roles>
  </group-specification>
</structural-specification>

```

```
<functional-specification>
  <scheme id="hw_choreography">
    <goal id="show_message">
      <plan operator="sequence">
        <goal id="show_w1"/>
        <goal id="show_w2"/>
      </plan>
    </goal>

    <mission id="mission1"  min="1" max="1"> <goal id="show_w1"/> </mission>
    <mission id="mission2"  min="1" max="1"> <goal id="show_w2"/> </mission>
  </scheme>
</functional-specification>

<normative-specification>
  <norm id="norm1"  type="permission" role="greeter" mission="mission1"/>
  <norm id="norm2"  type="permission" role="greeter" mission="mission2"/>
</normative-specification>

</organisational-specification>
```

Agents play the role `greeter` and commit to missions to show their words (bob shows "Hello" and alice shows "World"). Each agent has its own mission/goal/-word to show. As shown in figure 3.4, a greeter is permitted to commit to any mission, but we do not want all agents to commit to all missions that they possibly can. To solve this, each agent has a belief for the mission it should commit to. These beliefs are `my_mission(mission1)` for bob and `my_mission(mission2)` for alice. The decision to commit to a mission is implemented by the following plan:

```
1  // when the organization gives me permission to
2  // commit to a mission M in scheme S,
3  // do that if it matches the belief my_mission
4  +permission(A,_,committed(A,M,S),_)
5     :  .my_name(A) &  // the permission is for me
6        my_mission(M)  // my mission is M
7     <- commitMission(M).
```

The symbol + in line 4 means "in the event of coming to believe ..."; the code after `:` is conditions on what the agent believes to be the current situation, which are required for the plan to be used; and the code after `<-` is "deeds" (such as actions to execute and goals to achieve). This plan is thus triggered by the addition of a belief that agent `A` has the permission to commit to mission `M` in scheme `S`. If the value of variable `M` in the agent belief `my_mission(M)` matches the permitted mission `M`, the plan is applicable for the event and the agent does the action of committing to mission `M`.

The leaf goals of the social scheme should be achieved by the agents, and so they have plans for that:

```
// when I have goal show_w1, create subgoal say(...)
+!show_w1  <- !say("Hello").
+!show_w2  <- !say("World").
```

```
12
13 +!say(M) <- writeMsg(M).
```

The symbols `+!` in line 10 can be read as “in the event of having a new goal...”. The code `!say(...)` on the same line creates a new subgoal. With regard to the `permission` belief, goals `show_w...` come from the organization. The organization informs the agents about the goals they have to pursue considering the current execution state of the scheme and the commitments of the agent. In this example, all agents participating in the organization have plans for all `show_w` goals; the agents have the know-how to show both words and which word they show depends on the missions they have committed to.

Briefly, agents have plans to react to events produced by the organization (new permissions and new goals) and do not need to explicitly coordinate among themselves through communication; that is, bob does not need to send a message to alice anymore. Neither is the environment required to support coordination.

The application file for this implementation of Hello-World is as follows:

```
1  mas hello_world {
2    agent bob : hwa.asl {
3      focus: room.board
4      roles: greeter in ghw   // initial role for bob
5      beliefs: my_mission(mission1) // initial belief
6    }
7    agent alice : hwa.asl {
8      focus: room.board
9      roles: greeter in ghw
10     beliefs: my_mission(mission2)
11   }
12   workspace room {
13     artifact board : tools.Blackboard
14   }
15   organisation greeting : org1.xml {
16     group ghw : gg {
17       responsible-for: shw
18     }
19     scheme shw : hw_choreography
20   }
21 }
```

As previously, this file has entries for agents and workspaces, but now an organization block is added. In line 19, an organization entity is created on the basis of the XML file that describes the type of groups and schemes available in the organization. One group entity is created in line 20 (identified by `ghw`), and one scheme entity is created in line 23 (identified by `shw`). Line 21 states that group `ghw` provides the agents for the execution of scheme `shw`. Lines 5 and 11 assign role `greeter` to our agents in group `ghw`.[2] Lines 6 and 12 add beliefs in the agents regarding the missions to which they should commit.

The execution of the application file (`.jcm`) happens as follows:

1. The workspace `room` and the artifact `board` are created.
2. The group `ghw` and scheme `shw` are created and linked (responsible-for).
3. Agents bob and alice are created and they join the workspace `room`.
4. The agents are assigned the role `greeter`.
5. By playing this role, they start believing
   `permission(bob,_,committed(bob,mission1,shw),_)`
   `permission(bob,_,committed(bob,mission2,shw),_)`
   `permission(alice,_,committed(alice,mission1,shw),_)`
   `permission(alice,_,committed(alice,mission2,shw),_)`.
6. The addition of these beliefs trigger their first plan, and they commit to their missions. A picture of the overall state of the system is shown in figure 3.5.
7. When the agents have committed to their missions, the scheme `shw` has enough agents for it to be carried out, and the goal `show_w1` can be finally pursued.
8. Agent bob, being committed to `mission1`, is informed that goal `show_w1` can be adopted, and it does so; the message "Hello" is written on the blackboard.
9. Agent alice is then told to achieve `show_w2`, and it does so; the message "World" is written on the blackboard.
10. The scheme is finished.

We can notice a coordinated behavior: the words are always shown in the correct order. Moreover, the coordination is implemented in the organization and not in the agents (there is no code in the agents to coordinate their individual actions so that the overall system behavior is as expected).

3.5 Bibliographical Notes

Several platforms and programming languages have been proposed to implement MAS since the seminal paper of Shoham that proposed AGENT0 in 1993 (Shoham 1993). AGENT0 was followed by some programming languages also inspired by the mentalistic notion of agency; some of them are Jason (Bordini et al. 2007), 2APL (Dastani 2008), GOAL (Hindriks 2009), and ASTRA (Collier et al. 2015). The BDI agent model also inspired some frameworks that extended some existing language to support the agent programming (e.g., Jadex [Pokahr et al. 2005] and JACK [Winikoff 2005]). However, agent programming is not based exclusively on

2. Agents may decide to play roles on their own, and this feature is discussed in chapter 8.

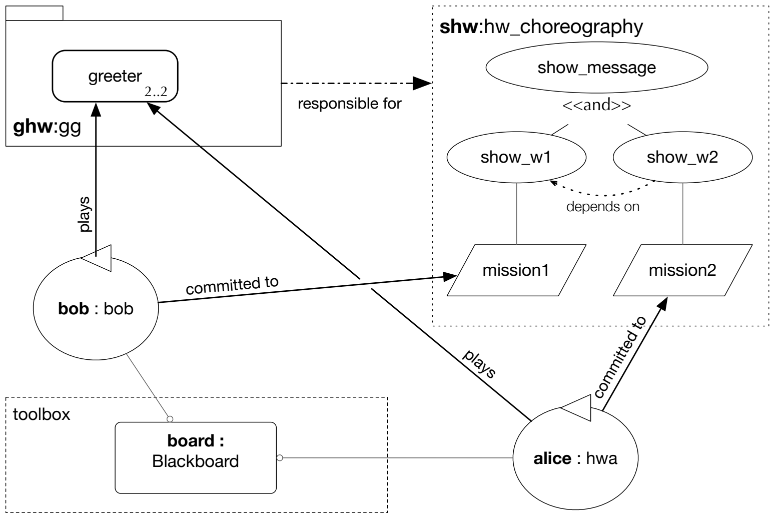


Figure 3.5
Coordination by organization: Entities of the Hello-World organization.

the BDI model. For instance, in the JADE popular platform, agents are programmed on the basis of a behavioral model (Bellifemine et al. 2007). Besides providing an agent programming model, most of the platforms also help the developer to launch agents in a distributed network, managing their life cycle and supporting their communication. These features usually follow the Foundation for Intelligent Physical Agents (FIPA) standards (Foundation for Intelligent Physical Agents 2000).

It should be noted that all cited tools are focused on the agent dimension. Some of them consider the environment as the network that provides a communication media (e.g., JADE) whereas others consider the environment to be external and include integration mechanisms (e.g., 2APL, GOAL, Jason, and ASTRA). The environment as well as the organization is not a first-class abstraction for them. It does not mean that other dimensions are not addressed by the community. For instance, the environment of an MAS can be implemented using the tools MASQ (Stratulat et al. 2009), GOLEM (Bromuri and Stathis 2008), and CArtAgO (Ricci et al. 2010). Options for the organization are MadKit (Gutknecht and Ferber 2000), KARMA (Pynadath and Tambe 2003), AMELI (Esteva et al. 2004), 2OPL (Dastani et al. 2009), ORA4MAS (Hübner et al. 2010), and THOMAS (Criado et al. 2011). However, these tools are not focused on the agent dimension and usually run on the basis of an ad hoc integration with an agent programming tool.

JaCaMo is the first fully functional platform that integrates the three dimensions as first-class abstractions. Because JaCaMo considers a mentalistic model of agency, it can align beliefs with observable properties or tell messages as well as obligations with goals. Although this characteristic constrains the way we program agents (we are somehow forced to program agents using BDI abstractions), it allows a synergistic integration between the dimensions. JADE, for instance, cannot implement the semantics of ACL messages because its agent model has no beliefs or goals. The programmer is responsible to properly code the interpretation of tell messages. In JaCaMo, messages are automatically interpreted: accepted tell messages become new beliefs for the receiver, for example. This platform has also served as the starting point for others, for example, JaCaMo+ implements interaction protocols based on commitments on top of artifacts (Baldoni et al. 2016).

A comprehensive review of tools and languages used to program MAS in general can be found in (Bordini et al. 2005; Aldewereld et al. 2016).

Exercises

Exercise 3.1 Change the implementation of the three versions of the `Hello-World` application: *coordination by communication* (in section 3.2), *coordination using the environment* (in section 3.3), *coordination by organization* (in section 3.4). These changes consist of

a) adding a new message with three words, "Hello Wonderful World," and a third agent to handle this new word;
b) printing the words of the message in reverse order;
c) printing the words of the message in parallel (hint: in the XML file where the organization specification is defined, options for the plan operator are `sequence`, `choice`, and `parallel`); and
d) printing the message again and again, as soon as it is printed.

For each of these changes, evaluate which version (coordination by agent communication, coordination using the environment, or coordination by organization) is easiest to implement.

Exercise 3.2 Change the agents so that they are capable of printing the "Hello World" message in different languages, and implement a mechanism to easily change the language.

Exercise 3.3 In the normative specification of the organization, replace `permission` with `obligation`, and notice the difference in the execution.

Exercise 3.4 In the structural specification, create two roles (considering the printing of "Hello World" as done in this chapter) or three roles (considering the printing of "Hello Wonderful World" as done in the first exercise), one for each word of the message.

4 The Agent Dimension

In this chapter, we delve into the details of the first MAOP dimension we consider in this book, the *agent dimension*. We start by recalling the overall picture of this dimension given in the preceding chapter. We also aim to convey the importance of this dimension within the context of multi-agent oriented programming. Then we look at each single programming concept and abstraction related to this dimension in the JaCaMo framework, noting that most of these concepts are used more generally in MAOP. After that, we discuss the agent execution model, explaining further agent-related concepts. We finish the chapter with further notes, including some historical notes for the reader interested in tracing the developments leading to the current state of the art.

4.1 Overview

To help us recall the overall picture about *agents* and the main concepts in this dimension, we take from figure 2.3 only the agent dimension and show it in figure 4.1. The approach we take in programming agents is based on the *BDI architecture* (see section 4.4 for references on this agent architecture).

agents

BDI architecture

Essentially, agents have *beliefs* about the current state of the environment, as well as beliefs about other agents and the state of the organization, and have *goals* that represent future states of the environment that are desirable to the agent (and possibly to its designer). On the basis of these (both the information about the present state and the representation of the states it would like to reach), the agent reasons in order to make decisions about the best actions to take in order to achieve those desirable states of affair. An *action* normally changes the state of the environment in which the agents are situated in some predetermined way.

beliefs

goals

action

One particular type of action, also shown in figure 4.1, that agents can take is a *communicative action*, that is, an action that allows an agent to directly communicate to one or more, possibly all, other agents in a multi-agent system. Communication in MAOP is based on the speech act theory (we give references to this in section 4.4).

communicative action

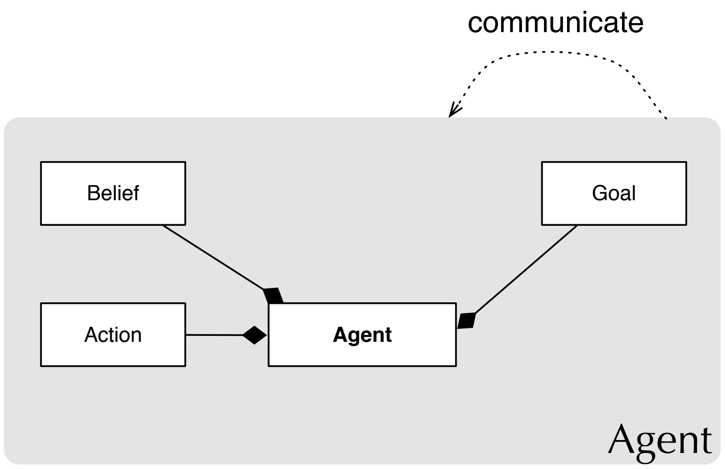


Figure 4.1
Main concepts in the agent dimension.

In practice, this means that when an agent sends a message to another, in addition to the actual content representing some knowledge, preference, or know-how, there is an explicit representation of the intended purpose of the message for the agent sending the message, which in turn allows the receiving agent(s) to know what to do with that message content. The intended purpose of the message is expressed by means of a *performative verb*, that is, a word such as *tell*, *achieve*, or *ask* that will affect what the receiving agent does with the content of the message.

performative verb

For example, if an agent receives a message with its contents expressing a property *finished* about a *cake* in a bakery, the consequences for the receiving agent will vary depending on the performative associated with the message. If the performative is "tell," the receiving agent will know that the sending agent wanted it to believe that the cake has been finished. If the performative is "ask" instead, then the receiving agent will know that the sending agent wanted it to answer whether the cake has been finished. If the performative was "achieve," then the receiving agent will know that the sending agent wanted it to take action in order to get the cake finished.

Of course, in a system with multiple autonomous agents, much further support for agent interaction is needed. The organization dimension covers some of it, and in chapter 9 we see in practice how this can be programmed. We also discuss further topics such as argumentation in chapter 11.

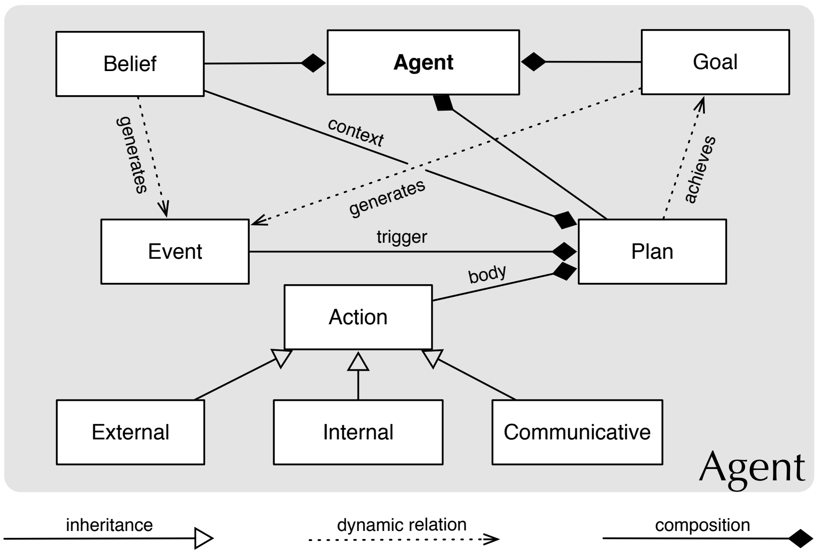


Figure 4.2
Concepts in the agent dimension.

4.2 Agent Abstractions

We now expand figure 4.1 to show all the concepts of this dimension in figure 4.2. In the next section we cover concepts related to the execution of an agent program, that is, structures specific to runtime rather than abstractions used to program autonomous agents in the approach used in this book.

The information that an agent currently holds about its environment, including other agents in it and the agent organizations, is represented as a set of *beliefs*. There is nothing special about *beliefs* in comparison with the way that information is represented in other programming languages.[1] The term *belief* is useful to remind us that agents typically have information that might be incorrect with regard to the actual state of the environment; for example, in the case that the areas of applications involve environments that are constantly changing, often in unpredictable ways, or in the case that obtaining such information (e.g., with sensors) can be faulty or inaccurate. Besides, agents typically do not have access to all the currently available information about the shared environment. In summary, the term *belief* reminds us that agents have to deal with inaccurate and incomplete information.

1. Note that the approach in this book uses a style of information representation similar to that of logic programming languages in particular, but not necessarily similar to that of other agent programming languages.

In the programming style used for the agent dimension in JaCaMo, we write beliefs using literals as is usual in logic-based programming, except that we also have annotations enclosed in square brackets. These can be used to store metainformation about the belief, including a special annotation named `source`, which JaCaMo uses to keep track of the origin of that information (e.g., the name of an agent, a `percept`, or `self` in the case of *mental notes* created by the agent itself). For example, if an agent has in its belief base the following belief:

mental notes

```
finished(cake)[source(percept)]
```

it means that the agent currently believes that the cake has been finished because it noted so by observing the environment. Note that instead of the term `cake` to refer to a specific cake, a variable could have been used. Variables are denoted by identifiers starting with an uppercase letter (e.g., `SomeCake`).

Whereas a belief states something about the current state of affairs, goals denote properties of the shared environment that the agent would like to become true and therefore does not currently believe to be true. Explicit representations of *goals* are of fundamental importance in *proactive behavior*: they lead the agent into action, including communication with other agents, so as to achieve a different state of affairs. In order to do so, an agent needs a plan of action (the *plan* concept is subsequently discussed). In other words, a plan gives a recipe for action that might lead the agent to achieve one of its goals. We can also use the bakery scenario to exemplify a goal. Suppose we did not believe the cake has been finished. In that case, we might currently wish it to be finished. In our language, goals are expressed similarly to beliefs except that they are preceded by an exclamation point `!`, and the source of a goal tells us the agent that delegated that goal. Therefore, if in our example we want to get the cake finished because a pastry chef named *John* asked us to get it finished, that could be represented as

proactive behavior

```
!finished(cake)[source(john)]
```

reactive behavior

As well as proactive behavior, we also want our agent to display *reactive behavior*. If an agent is pursuing some goal on our behalf, we will need the agent to be attentive to what is happening in the environment, because the action of other agents in the environment might prevent an agent from or indeed help an agent in achieving its goals. Circumventing problems and taking advantage of new opportunities is paramount for an agent to display intelligent behavior. An *event* represents either of two different kinds of things: changes in an agent's *goals* or changes in an agent's *beliefs*. The former is associated with proactive behavior, whereas changes in beliefs are important for reactive behavior. Note also that we are discussing *changes*, so that an event reflects, for example, that an agent has a new goal to achieve, or

event

that it no longer holds a particular belief. It is such changes (additions or deletions) that actually lead an agent to execute a plan. As an example, the event

```
-finished(cake)
```

means that the agent no longer believes the cake is finished (for example, because the agent realized some piece of decoration has fallen off the cake and thus may wish to act upon that event), whereas the event

```
+!finished(cake)
```

means that the agent has just adopted a new goal to reach a state in which the agent will believe that the cake has been finished properly.

Declarative vs. Procedural Use of Achievement Goals

Normally, achievement goals should be programmed *declaratively*; that is, a goal refers to a fact that the agent currently believes is not true, and if the goal is achieved, the agent will believe that the associated proposition has become true. For example, the `!finished(cake)` goal has been used declaratively. The agent has the goal to finish the cake because it does not believe that the cake is finished yet. After doing all the work, it is hoped, the agent will look at the cake and see that it is now completely finished, and so it will come to believe `finished(cake)`. However, our MAOP approach does not require goals to be programmed that way. We can also use them in a *procedural way* as simply a name for a sequence of actions we would like the agent to execute. For example, suppose that we have a procedure to get cream whipped, and the chef just wants to get an agent to follow that procedure; if the cream is not properly whipped at the end, it will fix it some other way. If the agent has a plan for `!whip_cream`, the chef could simply delegate that achievement goal to the assisting agent. When all the actions have been executed, the requested job will have been finished, regardless of the achieved results.

A *plan* is a recipe for action, in particular a recipe that allows an agent to achieve one of the possible goals that it might have or that allows it to react to opportunities or potential problems perceived from the state of the shared environment. That is, a plan deals with a specific type of *event* that is relevant to the agent. The set of plans an agent has at a particular moment constitutes its *know-how*. Syntactically, a plan is structured in three parts: *plan* *know-how*

Trigger The *trigger* or *triggering event* of a plan explicitly states which goal or reaction the plan is about. The software architecture underlying an autonomous agent keeps track of *events*, that is, of new goals the agent has to achieve (because, for example, another plan requires it or another agent requested it) and new beliefs that require the agent to react. In tackling outstanding events as recorded in the appropriate structure of the agent architecture, the *relevant plans* to consider are the plans that have a matching trigger. For example, agent John has a plan to handle the fact that he has a new instance of the goal to get a cake finished, and the bakery assistant agents have plans to react when they perceive that John is trying to finish a particular cake. The trigger part of these plans would be written as follows:

relevant plans

```
+!finish(cake) : ...

+finishing(john, cake) : ...
```

Context There may be many different plans to handle the same event. For example, to finish a cake one might get the icing and decoration ready and then finish the cake, but perhaps a healthier option would be just to spread a little whipped cream on top to follow a simple cake style instead. Different plans for the same purpose are *applicable* under different circumstances. That is what the *context* of the plan tells us. It is typically formed of conjunctions of beliefs that the agent is required to have at the time of choosing a plan of action for an event:

applicable
context

```
+!finished(cake)
  :  order(cake)[source(Client)] & lifestyle(Client, healthy)
  <- ...

+!finished(cake)
  :  is_for(cake, wedding)
  <- ...
```

Body The *body* of the plan is the recipe for action. Normally, an *action* is something an agent can perform in order to change the state of the environment (we subsequently present more about actions). The plan body stipulates the actions the agent is expected to carry out in order to handle the event that triggered the plan (such as the new goal instance or the newly acquired belief). Furthermore, complex plans might require the agent to achieve (other) specific goals amid the actions to be taken. Before we look at the various types of actions that can appear in a plan, we conclude the examples started above. The plans for John are

```
+!finished(cake)
  :  order(cake)[source(Client)] & lifestyle(Client,healthy)
  <- whip(cream);
     spread(cream,top);
     !have(decoration);
     decorate.
```

```
+!finished(cake)
  :  is_for(cake, wedding)
  <- !have(marzipan);
     cover(cake,marzipan);
     !piping_decorated(cake).
```

The first plan involves the agent executing two actions one after the other (whipping and spreading) to put some whipped cream only on top of the cake. Then the agent will need another plan to achieve the goal of getting the required decoration, which in turn could involve, for example, crushing nuts. Finally, the chef carries out the decorating action (assuming this one specific action that the agent can directly execute). The second plan is used for a prettier-looking cake appropriate for more formal events. It involves covering the whole cake with marzipan before carrying out the typically complex plan using piping techniques in order to decorate the cake.

The assistants, in their turn, have the following plan (which is to react to an observed event and is always applicable) to adopt the goal of assisting the pastry chef in finishing the cake:

```
+finishing(john, cake) <- !assist_finishing(cake).
```

Having this goal might lead those agents to act, for example, in such a way as to prepare or bring the marzipan, icing, nuts, and so forth, to the table where the chef is working. Of course, the different tasks need to be coordinated because, for example, some forms of icing cannot be prepared too long in advance of the actual decoration because it would dry out. In a subsequent section of this book, we look at how best to achieve such coordination.

As mentioned previously, an *action* is something an agent can execute; an action has one of the following three types:

External This is what we normally think of as an agent action. An *external action* is executed by an agent's *effectors* or *actuators*, that is, the specific means that agents has for changing the environment in which it is situated. For example, if the agent is controlling a robot with an arm, moving that arm, as implemented by the underlying control software, is an external action. The term *external* emphasizes that this type of action is implemented outside of the agent reasoning; the agent reasoning is precisely about which such actions will allow the agent to achieve its goals.

effectors
actuators

Internal An *internal action*, on the contrary, is implemented within the agent architecture, and it allows the agent to (atomically) run some available piece of code as part of its reasoning cycle. For example, if at some point the agent needs to run some legacy code or to run a piece of code that is better written in traditional programming languages (e.g., some image processing code), then an internal action is called for. However, because those pieces of code are run atomically, care should be taken for such actions not to block the reasoning cycle (described in the next section); otherwise, the agent ability to react to changes and carry out various other plans of actions concurrently is compromised.

Although internal actions refer to any code run within an agent's reasoning cycle, there is a very important type of internal action that is used to change its *internal state*—more specifically, to change the mental attitudes that determine the *mental state* of an autonomous agent. For example, there are internal actions that can be used to make an agent drop a particular goal it is trying to achieve or indeed to inspect which current goals the agent is pursuing. Because these are used for advanced MAOP, we do not exemplify them in this chapter.

mental state

Communicative We can also separately describe the *communicative actions*, which allow agents to directly communicate with each other. As discussed in the previous section, these actions are used to send messages to other agents. Inspired by speech act theory, such messages explicitly represent three different things: (1) the agent or agents that should receive the message; (2) the illocutionary force of the speech act, explicitly denoted by a *performative verb* such as "tell" or "achieve"; and (3) the actual content of the message, such as some piece of knowledge or know-how.

There are a few other programming constructs worth mentioning at this point. Besides achievement goals, the programming language also has *test goals*, which are denoted by prefixing it with `?` rather than `!`. That construct is used to retrieve information from the belief base in the body of a plan, so that the most up-to-date information is retrieved. For example, suppose that we need to retrieve what we believe to be the current phone number of the client whose cake we just finished; then we could have a plan like this one:

test goals

```
+!finished(cake)
  :  order(cake)[source(Client)]
  <- ...  ;
     ?phone(Client, Number);
     ...  .
```

The code line starting with `?` consults the current state of the belief base of the agent to try to find a number for the particular client that is instantiated to variable `Client`. If that information is not available in the belief base, there might be a plan telling the agent how to obtain such information. If we want to give that know-how to our agent, we can write a plan starting like this:

```
+?phone(Client, Number) : ... <- ...  .
```

Another important feature is to be able to reason about beliefs. For that purpose, there can also be *inference rules* which are similar to Prolog rules for reasoning about

inference rules

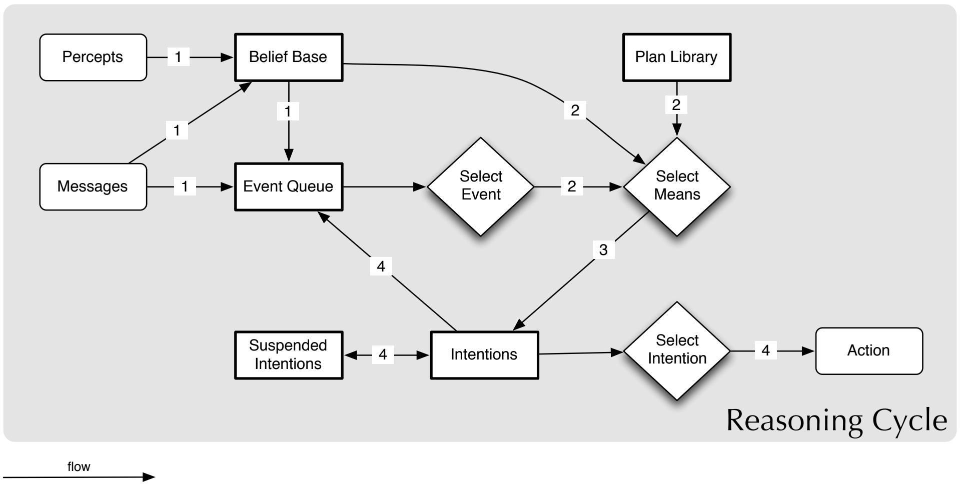


Figure 4.3
Reasoning cycle—runtime concepts in the agent dimension.

belief in the belief base, along with plain facts that are believed to be true by the agent. For example, suppose that if we know that a cake does not contain any animal products or products from companies that harm animals, then we can conclude that it is a vegan cake. We could add to the agent beliefs the following rule:

```
vegan(Cake) :- has_no_animal_products(Cake) &
               not (uses(Cake,Product) & produced_by(Product,Co) &
                   animal_harming(Co)).
```

Note that `has_no_animal_products(Cake)` itself could be a conclusion from another rule that checks the entire list of ingredients, for example.

4.3 Agent Execution

We now turn to the way an agent behavior is determined at runtime. Agents repeatedly go through *reasoning cycles* that start with perceiving the state of the shared environment and end with the choice of a particular action to be executed, possibly changing the state of the environment. Figure 4.3 shows the main steps of an agent reasoning cycle.

reasoning cycles

Before describing the four main steps of such a reasoning cycle, we briefly go through the definition of the main concepts shown in that figure and a few other related concepts.

A *percept* is a symbolically represented piece of information about the state of the shared environment. In real-world environments, percepts are typically obtained

percept

from sensors or cameras. These directly impact the agent's *beliefs*, as evident in our subsequent discussion of the reasoning cycle.

message

A *message* is a piece of communication received (asynchronously) from other agents. As discussed previously, messages can be used to inform an agent of something, to ask an agent to do something, or to provide an agent with a plan that will enable it to carry something out. Like many other aspects of an agent's reasoning cycle, there is a function that can be defined for each agent so that it selects only appropriate messages (e.g., messages from sources that the agent can trust or messages from sources that have appropriate authority). Further details are given by Bordini et al. (2007, chapter 7).

belief base

The *belief base* is a structure that keeps track of all information currently available to an agent. That can include information about the state of the environment (either perceived by the agent itself or informed by other agents), information about other agents, and things the agent wanted to remember about itself in previous internal states. All beliefs in the belief base are annotated with a label that explicitly states

source

the *source* of that belief. The source can be a percept in the case of belief originating from information gathered by the agent from its own sensors. Beliefs created from communication with other agents are annotated with the name of the agent that sent the message. Finally, the source can be the agent itself (something like the *mental notes* mentioned previously). Annotations on beliefs can also be used for various purposes, as shown in the more advanced examples in this book. An agent program typically has at least a few beliefs and a few plans. The latter gives the

initial know-how

initial state of the agent belief base

agent *initial know-how*, whereas the beliefs in the code form the *initial state of the agent belief base* (that is, the beliefs the agent will have when it first starts to run).

event queue

The *event queue* keeps track of all events that have effectively taken place. Recall that an event denotes some *change* in an agent's mental state. For example, the agent has a new goal to achieve, or a belief that arose from a percept that held true in the previous reasoning cycle no longer holds so that belief has been automatically deleted (these are just examples of possible events that might be in the event queue).

plan library

The *plan library* keeps all the know-how of an agent. It is a collection of plans—often various plans for the same triggering event. As explained above, to achieve the same goal, for example, one might have different plans that are used in different contexts. Recall that the agent program provides not only the initial state of the belief base but also of the plan library. However, note that agents can change their know-how during their execution. Whereas beliefs typically change constantly, plans tend not to change so much; however, changing the plans in a an agent's plan library is not only possible but is rather easy to do. Agents can exchange plans through communication, or AI planning techniques can be used for new plans to be created by the agent (see the references in section 4.4).

Given a particular event, a plan is said to be *relevant* if its triggering event matches that particular event. A plan is called *applicable* if its context holds true given the agent's current beliefs. Often, an agent might have various applicable plans for a given event. We saw earlier an example in which there were two different plans to finish a cake. If you consider those plans again, note how both might be applicable if a client with a healthy lifestyle is getting married. Note also that both plans get the same goal achieved—having the cake finished—but in different ways. Clearly, for a particular cake, only one of those plans has to be chosen for execution. Like beliefs, plans too can have annotations, and one use for them is to help select a plan when various different plans are all applicable to a current context.

relevant

applicable

When the agent chooses one applicable plan to handle a particular event, a copy of that plan from the plan library becomes an *intended means*. That is, if the event is to achieve a new goal, the agent is committed to doing so by following that recipe for action that is expressed in that intended means.

intended means

All intended means selected by the agent go into the *set of intentions*. An *intention* is a stack of intended means because a plan might require a (sub)goal to be achieved before further action is taken, and this dependency between some intended means must also be kept. Each separate intention in the set of intentions represents a different focus of attention for the agent. For example, agent John might be rolling out the marzipan when a timer goes off to say some pastry might be ready soon, so he will be keeping an eye on the oven while rolling out the marzipan.

intention

As discussed, an *action* is something an agent is capable of doing in order to change the state of the shared environment. Often, actions are associated with *operations* available in artifacts in the environment model or implemented by effectors (for example, through mechanical means available in a robot). From the agent reasoning point of view, actions to be executed are determined directly from the agent's current set of intentions.

The agent architecture uses three user-defined *selection functions*. In fact, default implementations for them are made available, but it is possible that domain-specific functions are required for particular applications. The three selection functions are as follows:

selection functions

Event The *event selection function* chooses one particular event from the event queue to be processed at each reasoning cycle. Normally, the selection function simply implements a FIFO policy, hence the *event queue* name. However, particular agents might need, for example, to prioritize certain specific events, and in this case a user-defined selection function is called for.

event selection function

means selection function
option selection function

Means The *means selection function* (often called *option selection function*) chooses one particular plan to be executed when multiple applicable plans are available to an agent at a given moment. Normally, the first applicable plan, in the order they appear in the plan library, is used; however, a particular agent might, for example, want to reason about properties annotated in plans in order to choose a plan that has a greater chance of succeeding or that typically has higher payoffs.

intention selection function

Intention The *intention selection function* chooses one particular intention among the available ones in the set of intentions (recall that each separate intention refers to a different focus of attention of the agent in regard to its environment, including other agents). This in turn determines which action will be taken at that particular reasoning cycle. Normally, this function operates a round-robin policy, that is, gives a fair chance for all intentions to execute actions. Again, particular agents might require, for example, sophisticated forms of reasoning about the relationships among intentions to ensure that they are scheduled efficiently.

suspended

Intentions may become *suspended* for various reasons. For example, actions are carried out by the agent's effectors in a different part of the agent architecture from that of the agent's reasoner. Therefore, after an intention requests an action execution, and before the action execution request is confirmed, the intention becomes suspended in the sense that further actions of that intention temporarily cannot be chosen for execution. Also, while an intention is waiting for a plan to be selected for a subgoal to be achieved, it must not be selected for execution, and so it is temporarily suspended.

We now turn to a high-level description of an agent reasoning cycle, as depicted in figure 4.3. The following description of the reasoning cycle process flow refers to that figure. Each reasoning cycle starts with retrieving all available information coming from the environment and ends with the selection of one action to be executed. The structures that interface between the agent reasoner and other components of the agent architecture (e.g., sensors and actuators) are shown as rounded boxes. The other main data structures are shown as boxes with a thicker frame. Selection functions are shown as diamonds. By grouping related parts of the reasoning process, we can divide the cycle into four main steps as follows:

1. Each reasoning cycle starts by retrieving from the part of the agent architecture in charge of interfacing with the shared environment the current set of available percepts as well as messages arriving from other agents. The percepts directly impact the agent's *beliefs*, whereas messages can affect both the set of beliefs as well as the agent's *goals* and *plans*. Recall that all changes in beliefs or goals are represented as events and placed in the event queue. In the agent's belief base, the beliefs that originated from percepts in the previous reasoning

cycle are explicitly annotated as such. This allows the agent architecture to determine the *changes* in perceptual information.

2. In this part of the reasoning, the event selection function is used to select a single event, for which all plans with matching triggering events are retrieved from the plan library. With that set of *relevant plans*, we need to check the context of each plan against the current state of the belief base, to determine the set of *applicable plans*.
3. At this stage, the *means selection function* is called with the set of applicable plans. Its output is the particular intended means chosen by the agent to handle the event selected in step 2. This needs to go to the set of intentions either as a new intention (if this plan is starting a new focus of attention for the agent, for example, because it is reacting to some newly perceived change in the environment or because it is adopting a goal requested by another agent) or as part of an existing intention in the case that the selected plan is to achieve a goal required to be achieved as part of another plan.
4. Finally, the intention selection function selects one particular intention, and the next required action of that intention is selected for execution in that reasoning cycle. So intentions directly determine the actions to be executed. Recall that there are three different types of actions that may appear in a plan, and in fact the part of a plan selected for execution may not be exactly an action but, for example, a new goal that the agent needs to adopt or the addition of a belief (or mental note) about something the agent will later need to remember, and so the belief base and event queue might need to be updated as part of that step. Furthermore, an intention may be suspended (as explained previously), and in all cases the set of intentions needs to be updated: an intention may be moved to the suspended set or the event queue to wait for a plan for a goal, and if it is kept in the intention set, at least the fact that one more action of the active plan has been executed needs to be updated in that intention.

4.4 Bibliographical Notes

The seminal work behind the BDI agent model and architecture is Bratman et al. (1988), with philosophical influences from Dennett's intentional stance (1987) and Bratman's work on the importance of the notion of *intention* for human practical reasoning (1987). An important addition to the original BDI agent model was Georgeff and Lansky's reactive planning (1987), which led to PRS (Georgeff and Ingrand 1989), the first practical implementation of a BDI agent platform. On the more formal aspects, in particular modal logics for agent mental attitudes, there is the seminal work by Cohen and Levesque (1990), and important work by Wooldridge (2000), Singh (1991), and many others. In fact, much has been written about the BDI model of agency and its roots in the philosophical literature; further details are given by Wooldridge (2009).

Some initial ideas toward agent-oriented programming were introduced by Shoham (1993). Perhaps the most influential early agent programming languages were Concurrent MetateM (Fisher 1993), ConGOLOG (Lespérance et al. 1996), AgentSpeak(L) (Rao 1996), and 3APL (Hindriks et al. 1997). Specifically regarding Jason[2], which underlies the JaCaMo platform used here, see work by Bordini et al. (2007), and on an initial formal semantics, see work by Vieira et al. (2007). Some of the most used agent languages and platforms include Jadex (Pokahr et al. 2005), GOAL (Hindriks 2009), JADE (Bellifemine et al. 2007), SARL (Rodriguez et al. 2014), JACK (Busetta et al. 1999), and ASTRA (Russell et al. 2015). A detailed account of some of the best-known agent programming languages was given by Bordini et al. (2005, 2009).

There are many open problems related to BDI agent programming, for example, the intention progression problem discussed by Logan et al. (2017). A strand of interesting research to address such open problems concerns the combination of agent-oriented programming with various AI techniques, for example, the pioneering work in combining planning and agent programming done by Sardiña et al. (2006). Chapter 11 gives further pointers on this, and see also the vision put forward by Bordini et al. (2019).

Exercises

Exercise 4.1 This exercise is about reactive and proactive behavior. Write an agent that reacts to the belief of having perceived in the environment an advertisement for a type of cake it likes. It *reacts* to this belief by adopting a goal to have that cake. The *proactive* behavior should involve different ways of achieving that goal, for example, baking the cake following a particular recipe or buying the cake in a bakery. Both will be applicable in different circumstances (e.g., having flour or having money). To test your agent, create an initial belief about the perceived cake ad, so you do not have to worry about an environment model.

Exercise 4.2 Create a prolog-like rule for reasoning about beliefs that checks whether all the required conditions to bake a cake at home are believed to be true. Change the code for the preceding exercise so that the context part of the plan for baking the cake becomes simpler just by using this inference rule.

Exercise 4.3 The plan for baking the cake may involve a subgoal of following a recipe for the cake dough. Before that, the agent needs to retrieve a recipe it may know from its belief base. Write a plan triggered by a test goal to be used in case the agent does not already know a recipe for that type of cake.

2. The Jason platform is developed by Jomi Hübner and Rafael Bordini and is available at http://jason.sourceforge.net/.

5 The Environment Dimension

Following the introduction of the agent dimension in the preceding chapter, we now incrementally enrich our set of programming abstractions by considering the *environment dimension*, which is used in MAOP to model resources and tools used by agents as first-class concepts. First, we introduce the *artifact* concept, which is the basic brick that can be used to design modular, dynamic, and composable environments—like artifacts in human environments—discussing the corresponding programming abstractions available in JaCaMo. Then, we see how agents can be programmed in order to create, use, and observe artifacts, able eventually to shape their own environments according to their needs and goals. Finally, we describe how complex artifact-based environments can be structured with *workspaces*, a concept introduced to model the topology of the multi-agent system, possibly distributed over multiple nodes of the internet.

5.1 Overview

environment

The concept of *environment* is used in MAOP as a programming abstraction to model resources and tools that agents can create, share, and use to do their jobs. An effective analogy for understanding the viewpoint is given by human work environments (figure 5.1). In order to do their work, humans (agents) not only communicate but also exploit resources and tools available in the work environment, which is often fundamental in making their action either effective or not effective. In addition, the objective of their work is often represented by some *artifact* that they incrementally build, possibly cooperatively. Analogously, in MAOP we introduce an abstraction layer that makes it possible to encapsulate in an MAS those services and resources that are not properly modeled as cognitive agents, being then neither autonomous nor proactive (goal-oriented). A simple example is a blackboard. Another example is a database or a shared knowledge base.

Figure 5.1 shows again the key concepts about the environment dimension, as presented in chapter 2, following the Agents and Artifacts (A&A) conceptual

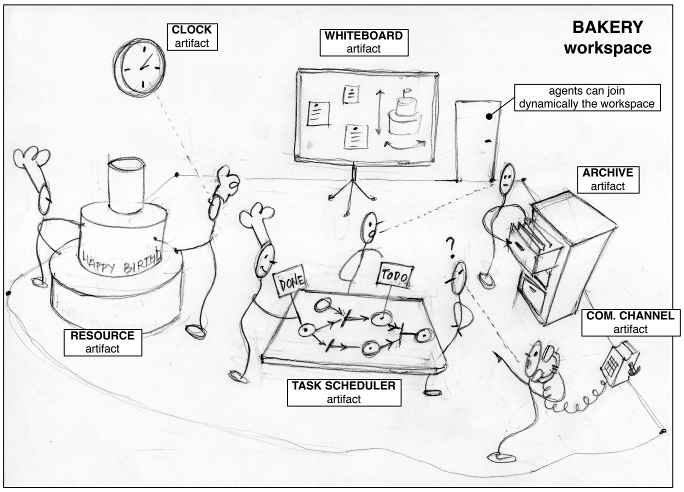


Figure 5.1
Agents and artifacts in the bakery workshop scenario.

model. In A&A, an environment is modeled as *workspaces* that agents can *join* to share and work with artifacts. The term *artifact* has been explicitly taken from activity theory and distributed cognition to recall the properties that such a notion has in the context of human environments. The *artifact abstraction* is introduced as a unit to structure and organize environments. From the MAS designer's point of view, artifacts are the basic building blocks—or rather the *first-class abstraction*—to design and engineer agent environments; from the agent point of view, artifacts are first-class entities of their world that they can instantiate, discover, share, and eventually use and observe to perform their activities and achieve their goals. If agents are the basic bricks to design the autonomous and goal/task-oriented part of the MAS, artifacts are the basic entities to organize the *nonautonomous, function-oriented*[1] (from the agent point of view) part of it.

artifact abstraction

Artifacts are then a natural abstraction to model and implement existing computational entities and mechanisms that are frequently introduced in the engineering of MAS representing shared resources, such as shared data stores, or coordination media, such as blackboards, tuple spaces, and event managers. Besides this, analogously to the human context, artifacts can be specifically designed to improve the way tasks get done by agents, by distributing actions across time (*precomputation*)

1. The term *function* is used here to mean *intended purpose*.

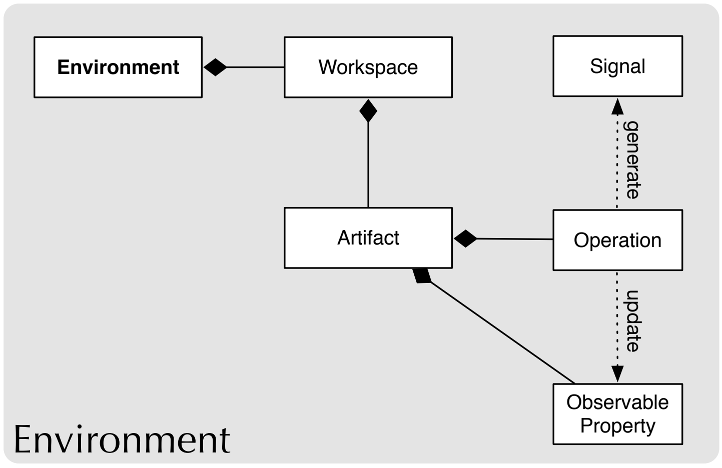


Figure 5.2
Main concepts in the environment dimension.

and agents (*distributed cognition*), and also by changing the way in which individuals perform the activity.

workspace abstraction

A&A also introduces the *workspace abstraction* in order to structure and organize the overall set of artifacts (and agents) in an MAS from a topological point of view. A workspace is a logic container of agents and artifacts, providing a logical notion of locality and situatedness for agents, by defining a scope for the interactions and observability of events as well as for the set of related activities carried by a group of agents using some set of artifacts. A complex MAS can then be organized as a set of workspaces, distributed among multiple nodes of the network, with agents possibly joining multiple workspaces at the same time.

As a summary, figure 5.3 shows the environment concepts presented so far in this chapter and relate them to the agent dimension presented in the preceding chapter. It is important to emphasize the generality of the A&A concepts. Although in this book we use the specific programming model and API provided by JaCaMo (as introduced from the next section), the notions of (autonomous) agents and (non-autonomous) artifacts sharing environment workspaces are quite general for the design and implementation of multi-agent systems.

5.2 Environment Abstractions

If the agent abstraction is meant to effectively model autonomous, proactive, goal-oriented computational entities, dually the artifact abstraction is meant to effectively model *function-oriented*, nonautonomous, computational entities that are

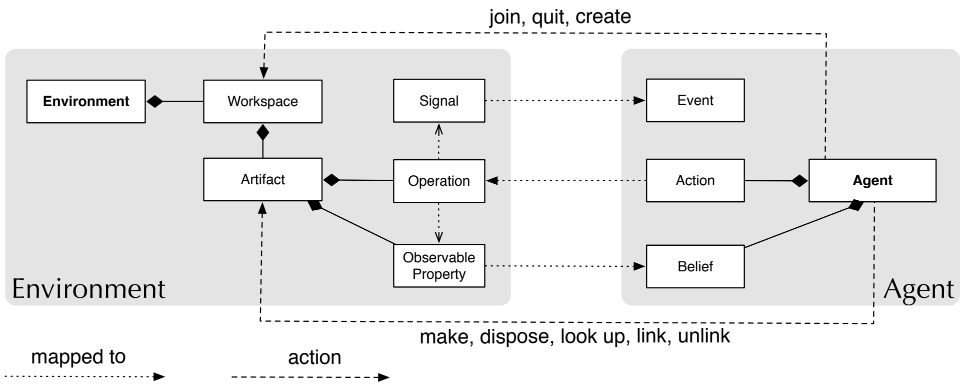


Figure 5.3
Zooming environment concepts map and their relationships with the agent dimension.

```
class Counter extends Artifact {

  void init(){
    defineObsProp("count",0);
  }

  @OPERATION void inc(){
    ObsProperty p = getObsProperty("count");
    p.updateValue(p.intValue() + 1);
    signal("tick");
  }
}
```

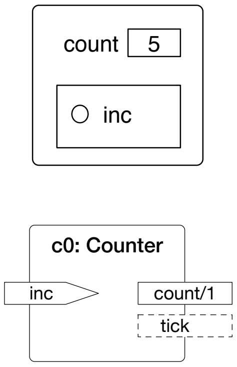


Figure 5.4
A simple `Counter` artifact example. On the left side is the Java source code of the artifact template class. On the right are two different representations—an abstract one (top) and a more detailed one (bottom)—used in the book.

.

meant to be *used* (observed or controlled) by agents for doing their jobs. Figure 5.4 shows the conceptual model proposed in A&A and adopted in JaCaMo to represent artifacts as computational entities. Artifact functionalities are defined in terms of *operations*, corresponding—from the agent point of view—to the actions provided to the agents using that artifact. Each operation is identified by a label and a list of input parameters.

From a programming point of view, similarly to objects in OOP, artifacts are instances of *artifact templates*. In JaCaMo, an artifact template can be implemented as a Java class extending a preexisting class—named `Artifact` and available in the API—and using a basic set of Java annotations and existing methods to define the elements of artifact structure and behavior. As as example, figure 5.4 shows the implementation of a simple counter artifact that keeps track of a count.

artifact templates

Operations are the basic units for structuring artifact functionalities and can either be atomic or a process involving a sequence of atomic computational steps. The `Counter` artifact has a single operation, called `inc`, to increment the count value.

operations

Observable properties represent the observable state of the artifact, which can be perceived by the agents observing that artifact. In our example, the `Counter` artifact has a single observable property called `count`, which keeps track of the count value. The value of the count property is incremented by the `inc` operation. Each time the observable state of an artifact is changed by an operation, an event corresponding to the new state can be perceived by agents observing the artifact. An artifact can have also a hidden, nonobservable state, which can be encoded for example in terms of private instance fields.

observable properties

In addition to observable properties, an artifact can generate *signals*, which are observable events that can be perceived by the agents using or observing the artifact. Signals can be used to represent any situation, condition, or message and are not necessarily related to observable properties, to be signaled to observing agents. In the example, the `Counter` artifact generates a signal called `tick` each time that the count is incremented.

signals

Besides observable properties and signals, an agent can receive feedback from an artifact in terms of output parameters in operations, that is, parameters whose value is intended to be set by the operation execution. From an agent point of view, these parameters represent *action feedback*. An example follows:

action feedback

```
class Calculator extends Artifact {

  void init(){
      defineObsProperty("last_result",0);
  }

  @OPERATION void add(double a, double b, OpFeedbackParam<Double> result){
    double res = a + b;
    getObsProperty("last_result").updateValue(res);
    result.set(res);
  }
}
```

The `add` operation in the `Calculator` artifact has an output parameter `result` that is set by the operation as the sum of the two operands `a` and `b`, passed as normal parameters. The parametrized class `OpFeedbackParam` is used to represent output parameters (which can be more than one for a single operation). In the example, a `last_result` observable property is also used to keep track of the result of the last operation executed. This is only a simple example of how action feedback and observable properties can be used together.

Finally, to support composition, artifacts can be *linked* together to enable inter-artifact interaction. This is carried out through *link interfaces*, which are analogous to interfaces of artifacts in the real world (for example, linking/connecting/plugging earphones into an MP3 player or using a remote control for a TV). Linking is also supported for artifacts belonging to distinct workspaces, possibly residing on different network nodes.

link interfaces

A Basic Taxonomy of Artifacts

Similar to objects and tools used by humans, artifacts can be classified according to the functionality that they provide. In the literature, three basic categories have been identified (Ricci et al. 2006):

- *Resource artifacts* This is the most general and common kind of artifact, representing some specific kind of resource to possibly be shared by agents. An example is the simple counter mentioned in this chapter; another is a shared knowledge-base artifact.
- *Coordination artifacts* These artifacts are specifically designed to provide coordination functionalities by enabling and managing in some way the interaction among agents. Examples range from synchronization mechanisms—analogous, for example, to barriers and semaphores in concurrent programming—to blackboards, auction machines, and workflow engines, up to artifacts used for organization management, which is discussed in chapter 8.
- *Boundary artifacts* These artifacts allow agents to interact with human users and, more generally, any actor or system that is external with respect to the MAS; an example is a GUI. Other examples can be found in chapter 10.

The design and implementation of coordination and boundary artifacts in particular may require the use of more advanced mechanisms provided by the artifact API to synchronize the execution of operations—analogously to condition variables in monitors—and manage the execution of asynchronous computations, interacting with external threads. These mechanisms are introduced in the next chapters.

Working with Artifacts from an Agent Perspective

The set of actions available to agents to work with artifacts can be categorized in three main groups:

1. actions to create/look up/dispose of artifacts;
2. actions to use artifacts, thereby triggering operations and observing properties and signals; and
3. actions to link/unlink artifacts.

Next, we describe these actions in more detail.

Creating and discovering artifacts We start with the *artifact creation and discovery* actions. Artifacts are meant to be created, discovered, and possibly disposed of by agents at runtime; this is a basic way in which the model supports dynamic extensibility (besides modularity) of the environment. In figure 5.5 (top), we show a simple example in which a *user* agent creates an artifact and makes use of it by performing operations, and an *observer* agent discovers the existence of that artifact and reacts to changes in the observable properties.

instantiating artifacts

The action `makeArtifact(Name,Template,Params,Id)` instantiates a new artifact called `Name` of type `Template` inside a workspace, getting as an action feedback its identifier `Id`. The logical name identifies the artifact inside a workspace. Artifacts belonging to different workspaces can have the same logical name, so in addition to the logical name, each artifact has a unique identifier generated by the system and returned as action feedback. As shown in figure 5.5 (left-hand side), in the plan for goal `!create_and_use`, an agent creates an artifact called `c0` as an instance of the artifact template `Counter`, passing as initial parameter the value 10. Dually to `makeArtifact`, `disposeArtifact(Id)` is used for removing an artifact from a workspace.

disposing artifacts

Artifact discovery concerns the possibility of retrieving the identifier of an artifact located in a workspace given either its logical name or its type description (i.e., the template used to create it). To this purpose, `lookupArtifact(Name,Id)` retrieves an artifact unique identifier given its logical name. In figure 5.5 (right-hand side), the plan for goal `!discover_and_observe` is used by another agent (different from the agent that created the artifact) to look for an artifact called `c0` and

looking up artifacts

observe it (by focusing on it). To that purpose, it first tries to retrieve the identifier of the artifact by means of the `!locate_count` subgoal, in which it repeatedly executes an artifact lookup until the artifact is found.

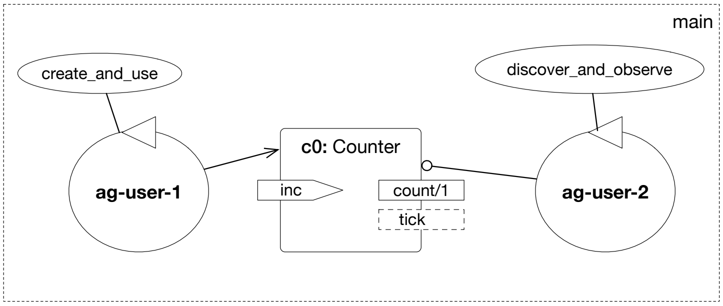
Executing operations on artifacts For agents, using an artifact essentially involves two aspects: being able to execute operations actually listed in the artifact usage interface, and being able to perceive artifact observable information in terms of observable properties and signals.

For the first aspect, from an agent point of view, artifact operations represent *external actions* provided to agents by the environment.[2] By executing an action `op(Params)`, the corresponding operation provided by an artifact in the workspace is triggered. As shown in figure 5.5 (left), a `user` agent creates an `Counter` artifact named `c0`, and then it increments it by performing an `inc` action that triggers the corresponding `inc` operation on the artifact. In fact, if more than one artifact in the workspaces an agent joined provides the operation, the identifier of the artifact target of the operation can be specified by means of an annotation `[artifact_id(Id)]` following the action. If no annotation is specified and there are multiple artifacts providing the operation, one is chosen (nondeterministically, from the perspective of the agent).

executing operations

The action executed by the agent succeeds if the corresponding operation completes with success; conversely, the action fails if either the specified operation is not currently included in the artifact usage interface or if some error occurred during operation execution, that is, the operation itself failed. By successfully completing its execution, an operation may generate some results that can be returned to the agent as action feedback. By performing an action, the enclosing agent intention is suspended until an event reporting the completion of the action (with either success or failure) is received. By receiving the action completion event, the action execution is completed and the related plan resumed. It is worth remarking that even if an intention is suspended, the agent is not blocked, and the agent reasoning cycle can go on processing percepts and executing actions related to other plans. Thus in artifact-based environments, the repertoire of external actions available to an agent is defined by the set of artifacts that currently populate the environment. This implies that the action repertoire can be dynamic, because the set of artifacts can be changed dynamically by agents themselves, instantiating new artifacts or disposing of existing artifacts.

2. See chapter 4 for the distinction between internal and external actions in agents.



```
/* user agent */

!create_and_use.

+!create_and_use : true <-
  makeArtifact("c0",
        "Counter",[10],Id);
  inc;
  inc [artifact_id(Id)].
```

```
/* observer agent */

!discover_and_observe.

+!discover_and_observe <-
  !locate_count(Id);
  focus(Id).

+count(V) <-
  println("count:  ",V).

+tick <- println("tick!").

+!locate_count(Id) <-
  lookupArtifact("c0",Id).
-!locate_count(Id) <-
  .wait(10);
  !locate_count(Id).
```

Figure 5.5
Agents using and observing an artifact. At the top is a diagram for the user and observer example. At the left is shown the `user` agent creating and using a `Counter` artifact called `c0`. At the right is shown the `observer` agent discovering the `c0` artifact, observing it, and reacting to changes to its `count` observable property.

Predefined Artifacts

Even the basic actions such as `makeArtifact` are like every other external action; that is, they are uniformly available to agents because there is an artifact featuring them as operations. In particular, every workspace is equipped by default with an artifact called `workspace` that provides core functionalities for creating, searching, and managing artifacts. Moreover, it provides a set of observable properties that make it possible for agents to know basic dynamic information about the workspace, such as the set of available artifacts (through beliefs of the form `artifact(Name, Template, Id)`). The `workspace` artifact is one of the predefined artifacts that are created by default in workspaces, as basic tools that agents can use in their activities. Other predefined artifacts include

- the `console` artifact, which provides operations to interact with the standard input/output (e.g., the `println` action used in the examples); and
- the `blackboard` artifact, which implements a simple (tuple-space based) blackboard, useful for agent coordination.

The complete description of these artifacts is part of the documentation available on the platform website.

observing artifacts

Observing artifacts Besides executing operations on artifacts, using them typically requires the capability of *observing* them. An agent can start perceiving observable properties and signals generated by an artifact by executing the `focus(Id,Filter)` action, specifying the identifier of the artifact to observe and optionally a filter to further select the subset of observable properties and signals in which the agent is interested. Dually to `focus`, the `stopFocus(Id)` action is provided for the case an agent no longer wants to observe a particular artifact.

In observing an artifact, observable properties of the artifact are mapped directly into beliefs in the agent's belief base. Every time an observable property on a focused artifact is updated, "behind the scenes" the artifact generates an event that is automatically perceived by the agent and the corresponding belief is updated, generating a belief change event.[3] This makes it possible to write plans that react to changes in observable properties. In figure 5.5 (right), an `observer` agent first discovers the `c0` counter, and then it starts observing the artifact and reacts every time the belief about the observable property `count(V)` is updated. An event is

3. Recall that changes in beliefs are one possible type of trigger for plan execution in agents.

generated also when the belief related to the observable property is created for the first time (i.e., when the focus action succeeds).

Signals, by contrast, are not related to observable properties; they are like messages generated on the artifact side that are asynchronously processed on the observing agent side. Accordingly, no beliefs about signals are kept by default.[4]

An agent can focus (observe) multiple artifacts at the same time, even of the same kind. To distinguish observable properties with the same name but from distinct artifacts, beliefs about observable properties are annotated with the specific identifier of the artifact that is the source for that belief. In particular, each belief about the observable property of an artifact is annotated with `artifact_id(Id)`, `artifact_name(Name)`, and `workspace(Id)`, carrying on information about the unique identifier of the artifact, its name, and the identifier of the workspace where it is hosted, respectively. These annotations can be used, for example, in writing agent plans:

```
+count(V) [artifact_name("counter")] <- ...
+count(V) [artifact_id(Id)] <- /* use the Id */...
```

Important remarks about the observation semantics:

- *Observation completeness* The model is such that no events can be lost. That is, for every observable state *s* generated by an artifact—that is, for every new value *v* made observable about an observable property *prop*—that state is perceived by every agent observing the artifact, and corresponding internal events are generated.
- *Event ordering* Observable states and events generated by an artifact are perceived by every agent observing the artifact in the same order in which they are generated. Conversely, no order is defined between events generated by different artifacts.
- *Atomic perception* If two observable properties are changed in the same operation execution, then their change is perceived through a single percept and the corresponding beliefs on the agent are updated in the same reasoning cycle, generating multiple internal events that can be processed in various subsequent reasoning cycles.

Linking artifacts Finally, we describe the *artifact linking* actions. Artifacts in human environments are often designed to be connected together, in order to combine their functionalities. Analogously, artifacts here can be *linked* together by agents to allow one artifact (the *linking* one) to execute operations over another

linking artifacts

4. Of course, occasionally an agent may want to remember that a particular signal occurred, which can easily be done by the agent by adding a particular belief to its belief base when reacting to the signal.

```
public class LinkingArtifact
      extends Artifact {

  private static final String
          linkedCount = "linkedCount";

  void init(){
    definePort(linkedCount);
  }

  @OPERATION
  void test(){
    execLinkedOp(linkedCount,"inc");
  }

  @OPERATION
  void test2(OpFeedbackParam<Integer> v){
    execLinkedOp(linkedCount,"getValue", v);
    log("back from linked op.: "+v.get());
  }
}
```

```
public class LinkableArtifact
      extends Artifact {

  int count;

  void init(){
    count = 0;
  }

  @LINK
  void inc(){
    count++;
  }

  @LINK
  void getValue(
      OpFeedbackParam<Integer> v){
    v.set(count);
  }
}
```

```
!test_link.

+!test_link
  <-  makeArtifact("myArtifact","LinkingArtifact",[],Id1);
      makeArtifact("count","LinkableArtifact",[],Id2);
      linkArtifacts(Id1,"linkedCount",Id2);
      println("artifacts linked: going to test");
      test;
      test2(V);
      println("value ",V).
```

Figure 5.6
Linking artifacts. On the left is an example of linking artifact. A port named `linkedCount` is defined and used to trigger the execution of operations on the linked artifact through operations `test` and `test2`. On the right is an example of a linkable artifact. The operations annotated with `@LINK` can be called by linking artifacts. At the bottom, an agent creates two artifacts, linking them together, and executing operations on the linking artifact.

artifact (the *linked* one, which should be linkable by exposing a proper link interface). To link together two artifacts, the action `linkArtifacts(LinkingArId, LinkedArId,Port)` is made available to agents.

On the linking artifact side, the notion of `Port` is used to (indirectly) refer to the linked artifact in the artifact code. Once artifacts have been linked together, the linking artifact can execute operations on the artifact attached to some `Port` by means of an `execLinkedOp(Port,OpName,OpArgs)` primitive. Figure 5.6 (left) shows an example of linking artifact defining a `linkedCount` port and using it in the `test` and `test2` operations to execute operations over the linked artifact.

On the linked artifact side, a *link interface* is exposed, defining the set of operations that can be executed by linking artifacts. A link interface is defined by annotating with @LINK those operations of an artifact that can be linked by other artifacts.[5] Figure 5.6 (right) shows an example of a linkable artifact exposing a couple of operations annotated with @LINK. The semantics of the link operation execution is the same as operations executed by agents: the operation request executed by the linking artifact is suspended until the operation on the linked artifact has been executed, whether with success or failure.

Figure 5.6 (bottom) shows an example of agent code creating and linking two artifacts (a `LinkingArtifact` and a `LinkableArtifact`) and then interacting with the linking artifact, indirectly executing operations also on the linked one.

Modularity—Encapsulation—Reusability

The environment model used in JaCaMo has many of the most important features expected of programming languages and software engineering methodologies. For example, artifacts provide a natural means for *modularity* because artifacts can relate to other artifacts through linked operations. An artifact is foremost a mechanism for *encapsulation*, because all the operations and observable properties conceptually related to an artifact entity perceivable by agents are implemented within the same artifact construct. Another important feature is *reusability*; needless to say, the same artifact template will be useful in many different systems that a developer may want to implement—think of a table, a blackboard, a coffee machine, or any other real-world artifact; and it will be easy to see how frequently the same artifacts are useful in many different contexts, and the same applies to artifacts in different multi-agent systems.

Structuring Artifacts into Workspaces

As mentioned in 5.1, in the environment model adopted in JaCaMo, artifacts are collected into *workspaces*, defining the topology of the environment. A workspace can be understood as a *logical place* containing artifacts, as well as a context of work for agents' activities. In order to access and use the artifacts of a workspace—that is, in order to share that context of work—an agent must first *join* it. An agent can join and work in multiple workspaces, and multiple agents can work in the same workspace concurrently.

By default, an MAS contains a single workspace. A complex environment, however, can be structured in terms of multiple workspaces, organized hierarchically,

5. The same method can be marked with both @LINK and @OPERATION to represent an operation that is part of both the usage and the link interface.

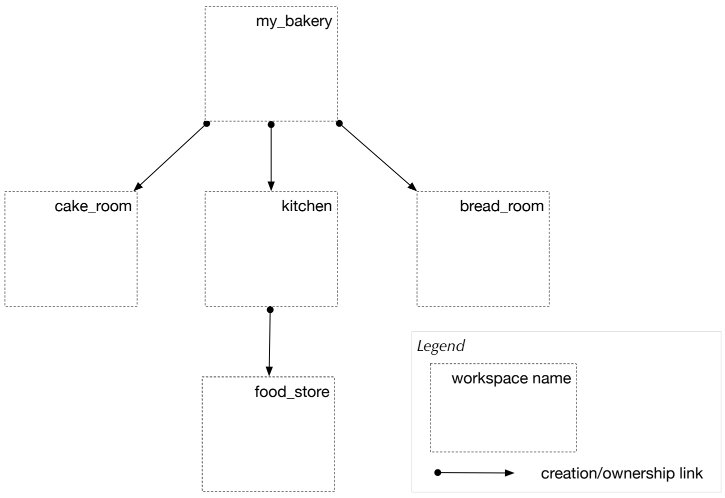


Figure 5.7
Overview of distributed environments.

similar to file systems; see figure 5.7. There is a root workspace called `main` by default, but more specific names can be used; for instance, in figure 5.7, the root workspace is called `my_bakery`. Each workspace can have one or more child workspaces but only one parent. As in file systems, logical paths can be used to refer to a workspace, for example, `/my_bakery/cake_room`.

On the agent side, some actions are available to work with workspaces. First, agents are spawned, or enter an MAS, in a specific workspace (their `home` workspace), which does not usually change during their lifetime within that MAS, except in the case of mobile agents. Once an MAS has been entered, an agent can work concurrently in multiple workspaces of the MAS by simply joining them.

joining a workspace

A `joinWorkspace` action can be used to join any workspace of the MAS by specifying the full path name of the workspace and getting its unique identifier as action feedback.[6] An example is `joinWorkspace("/my_bakery/cake_room",Id)`. Once a workspace is joined, the agent can interact with all the artifacts in that workspace. To exit a workspace, the `quitWorkspace(Id)` action is provided.

creating a workspace

Besides joining, an agent can *create* a new workspace by means of a `createWorkspace` action, specifying the identifier of the parent workspace and the logical name to be used for the new child workspace. Workspaces can be *disposed of* as well, by means of a `removeWorkspace` action, specifying the full

disposing of a workspace

6. The workspace identifier output parameter is optional.

name (path) of the workspace to be removed. Finally, to create an access link between two workspaces, there is the action `linkWorkspaces(From,To,Name)`, where `From` is the path of the workspace where the link is created, `To` is the path of the workspace to be linked, and `Name` is the label of the access link.

A simple example of agent source code working with workspaces is shown:

```
+!test_workspaces
  <- createWorkspace("/main/w0");
     joinWorkspace("/main/w0",W0);
     println("hello in ",W0);
     createWorkspace("w1");
     joinWorkspace("w1",W0);
     println("hello in ",W1).
```

The agent creates a couple of workspaces—`w0` and `w1`—and then joins them, printing a message on their `console` artifact (which is available by default in every workspace).

A couple of points deserve further attention and explanation. The first point concerns how the second workspace `w1` is referenced; as for file systems, a *relative* path is used instead of an absolute path—relative paths don't start with `/`. That is, analogously to the notion of the current directory in using operating system shells, there is in this case the notion of current workspace, which corresponds to the *last workspace joined* (in the case that the agent is working concurrently in multiple workspaces). Relative paths are solved with respect to the current workspace. In the example, when an agent in an MAS is booted, by default it joins the root workspace (called, by default, `main`). By creating and joining the `w0` workspace, `w0` becomes the current workspace. Then, the workspace `w1` is created as a child of `w0`, because a relative path `w1` is used to reference the workspace. This is equivalent to `createWorkspace("/main/w0/w1")`. It is worth noting that, like in file systems, `..` can be used to refer to the parent workspace, so `createWorkspace("../w1")` would have created the workspace inside `main`.

The second point, related to the first one, concerns the execution of operations over artifacts without specifying the artifact identifier (or the target workspace). In the example, the action `println` refers to the corresponding operation provided by the `console` artifact. A plausible question here could be "If the agent is working in multiple workspaces, which specific `console` artifact is used when the action `println` is requested?" The answer is that the current workspace is used; that is, when an action (operation) is specified without including the artifact identifier, the current workspace is implicitly considered to be the target, and then an artifact providing that operation is considered in that workspace. Therefore, in the example, the first `hello` message is printed by the `console` artifact in workspace `w0`, whereas the second `hello` message is printed by the `console` artifact in workspace `w1`. The current workspace is meant to represent the current context of work of an agent and, for this reason, it is bound to the current intention in execution; that is, each intention can have its own current workspace, and if an agent has multiple intentions in execution at the same time, the current workspace is automatically switched according to which intention is executing.

Finally, workspaces of the same MAS can be created and be in execution on different network nodes, yielding a distributed system. For instance, as shown in figure 5.7, `my_bakery`, `kitchen`, and `food_storage` workspaces could be running on some node, while `cake_room` and `bread_room` could be running on some other hosts. The same node can host the execution of multiple workspaces, whereas a workspace is in execution on a specific node (that is, a single workspace is not distributed). This topic is fully developed in chapter 7.

5.3 Environment Execution

On the agent side, control flows are needed to execute reasoning cycles, carrying on agent activities. On the environment side, control flows are needed to execute operations triggered inside artifacts. The execution model of operations inside artifacts is such that

- operations requested on different artifacts—either by different agents or the same agent—can be executed concurrently, possibly by different control flows; and
- operations requested on the same artifact are executed sequentially, enforcing mutual exclusion; that is, only one operation at a time can be in execution within an artifact.

If an agent requests the execution of an operation over an artifact where another operation (or the same one) is already in execution, the request is enqueued and served as soon as the current operation execution has finished or it has been suspended.[7] On the agent side, the execution of the corresponding (external) action is suspended, that is, the intention in execution is suspended (see section 4.3), until an action event corresponding to the completion of the operation (either with a success or a failure) is perceived. In fact, each time an operation in execution finishes, a corresponding action event (either completion with success or failure) is generated and delivered to the agent that triggered that operation.

The execution of an operation inside an artifact can change the observable state of the artifact and generate signals as well. An important point here concerns the semantics adopted to make such an observable state and related changes, as well as

7. More about suspended operations is provided in chapter 7.

signals, available to agents focusing the artifact as percepts. Such semantics follows the following principles (mentioned in a previous section):

- Inside an operation, it is possible to update multiple observable properties without necessarily making each single update observable. In that case, agents observing the artifact perceive a new observable state of the artifact in which possibly multiple properties have been changed.
- Each time a signal is generated, it is made immediately perceivable by observing agents, along with the current observable state of the artifact.
- The order of the changes—that is, sequence of observable states—occurring inside the artifact should be preserved when perceived by agents, and no change or event should be lost.

This semantics impacts the API used to implement artifacts as follows:

- Inside an operation, in updating an observable property (e.g., by calling an `updateValue` method), the observable property is updated but the new state is not made immediately observable to the agent.
- The new state is made observable at the end of the operation execution, or when a signal is generated (with the `signal` primitive), or by an explicit commit on the observable state. This can be accomplished by calling a specific primitive included in the artifact API, called `commitObsState()`.

Artifacts vs. Objects and Monitors

Artifacts have some similarities to objects in OOP. Like objects, artifacts can be used to model nonautonomous entities providing an interface to be used, composed by a set of operations. The agent-artifact interaction model, however, is profoundly different with respect to the model adopted in OOP to define the interaction between objects. In the case of objects, a method call implies a transfer of control between an object (calling method) and another object (called method), as in the case of procedure calls. That is, it is a synchronous call. In the agent-artifact case, control is encapsulated inside agents and cannot be transferred. Therefore an agent invoking an operation over an artifact—that is, executing an action—is not transferring the control; the execution of the triggered operation is carried out by another logical control flow, provided by the environment. On the agent side, the plan in execution is suspended until the action in execution is either completed or failed—it is notified by means of an event (an action event). Even if the plan is suspended, the agent reasoning cycle goes on because of encapsulation of control, so other intentions will continue to be pursued by the agent.

Another main difference with respect to objects is about observable properties. Objects in OOP conceptually do not have an observable state composed of properties; a good practice in OOP is to model every interaction with objects through a proper interface (that is, methods). Accordingly, instance fields in objects should be declared private, to enforce information hiding. This is the case also of artifacts, which can have a hidden private state. However, the advanced concept of observable properties in artifacts is strongly related to the agent abstraction as an entity acting and perceiving its environment, given that an environment has some observable state. This is directly captured in artifacts by observable properties. It is worth noting that observable properties in artifacts are not like object instance fields that are declared public. In fact, observable properties cannot be directly written; they can be modified only by artifact operations. Besides, changes to observable properties raise them automatically to observable events that are perceived by agents observing (that is, focusing their attention on) the artifact. The stream of events generated by changes to an artifact observable property is quite similar to the asynchronous streams in reactive programming.

Like monitors in concurrent programming, artifacts can be safely used by multiple agents concurrently—they are thread-safe by construction. In fact, as in monitors, in artifacts only one operation can be in execution at a time, so mutual exclusion is enforced. There could be multiple operations that have been triggered and are in execution; however, in that case all but one are suspended (for instance, in awaiting some condition). Monitors, too, have a different model for the interface. When a process invokes an entry on a monitor, the process control flow is transferred, as in the case of objects—that is, conceptually, the control flow executing entries inside monitors is one of calling processes. As mentioned previously, this is not the case for artifacts. Monitors can execute operations in other monitors, even though this an error-prone practice, that can easily lead to deadlock situations—in fact, when a monitor calls the entry of another monitor, the lock on the calling monitor is not released. Artifacts use linkability as a more disciplined way to model interactions between artifacts.

5.4 Bibliographical Notes

In the classical AI view (Russell and Norvig 2003), the notion of environment is used to identify the external world (with respect to the system, being a single agent or a set of agents) that is perceived and acted upon by the agents so as to fulfill their tasks. In agent and multi-agent system literature, the environment is a primary concept, to explicitly represent the computational or physical place where agents are situated, defining the notion of agent perception, action, and agent *goals*, defined in terms of states of the world that an agent aims to bring about. The idea of environment as a *first-class abstraction* for MAS engineering has grown up in the context of agent-oriented software engineering (Weyns et al. 2007), enhancing the classical view by seeing the environment as a suitable place to encapsulate functionalities and services to support agent activities. We refer the interested reader to Weyns and Parunak (2007) for a survey of the research developed in this context. Among that work, some explored the idea focusing on the *architectural* level; in that case, the architecture of MAS is extended with an environment layer, encapsulating some functionalities (Weyns and Holvoet 2006). Some others instead explored the idea down to the *programming* level (Ricci et al. 2010), which is the perspective adopted in this book.

A further important perspective that involves the environment dimension in MAS is given by the Environment Interface Standard (EIS) initiative (Behrens et al. 2011), which provides an effective support to integrate agents written in different agent technologies to work in the same application environment. EIS has been used, for instance, to integrate cognitive agents developed in the GOAL language (Hindriks 2009) as bots in the Unreal game environment (Hindriks and Dix 2014), and as the reference environment interface in the Multi-Agent Programming Contest (Behrens et al. 2012).

Exercises

Exercise 5.1 Extend the `Calculator` artifact introduced in section 5.2 as follows:

a) add an operation `sum(a: double, ?result: double)` that computes the sum of the argument passed `a` and the last result, updating accordingly the observable properties and the output parameter; and

b) add an operation to memorize the last result (`storeResult`) so as to recall it in the future by means of a `recall` operation, which is meant to reset the `lastResult` observable property with the stored result.

Write a JaCaMo program with an agent testing the calculator.

Exercise 5.2 Design and implement a `SharedDictionary` artifact that functions as a shared dictionary. The dictionary keeps track of information items identified by a keyword (a String) and a corresponding content (a Jason Term). The artifact should provide an operation for adding new information items and for retrieving an information item given the key.

Exercise 5.3 Extend the preceding artifact with the possibility of perceiving the content state of the dictionary as observable, so that agents can observe that state and potentially react to changes in it.

Exercise 5.4 Design and implement a `TicTacToeBoard` artifact for agents playing the tic-tac-toe game.

Exercise 5.5 Implement a full tic-tac-toe JaCaMo game in which two agents play the game using the `TicTacToeBoard` artifact.

6 Programming an Agent and Its Environment

In this chapter, we see in practice the programming of agents within an environment by considering a toy (but realistic) scenario referred to as a *smart room*. In this chapter, we proceed step by step to program a simple single-agent system controlling the temperature of a room. The case study is further extended with multiple agents situated in the same working environment, focusing on the programming of interaction among agents in chapter 7 and on the programming of agent organizations in chapter 9. Both chapters investigate how we can coordinate the work of the agents so that they produce a coherent and cooperative global behavior.

6.1 Programming a Proactive Smart Room

We start with a very simple JaCaMo program that has the objective of controlling the temperature of a room so that it reaches some desired value. Given the availability of multiple design and programming dimensions, we investigate here how to exploit them incrementally. For now we concentrate on the programming of autonomous agents and artifacts in the environment.

Designing the agent and the environment In order to design the environment and the agent operating in it, we apply the separation-of-concerns principle discussed in chapter 2. As shown in figure 6.1, we thus design our first JaCaMo program in terms of

- the control and decision-making activities needed for the system to autonomously achieve its objectives. We encapsulate them using the agent abstractions and define a `room_controller` agent. This agent has the initial simple goal of bringing the *temperature* of the room to some specific value; and
- the tools or resources that are necessary to realize these activities, i.e., tools and resources that the agents need to achieve their objectives. These tools and resources also include what is needed to interface and interact with the external environment. We use environment abstractions to model and program them:

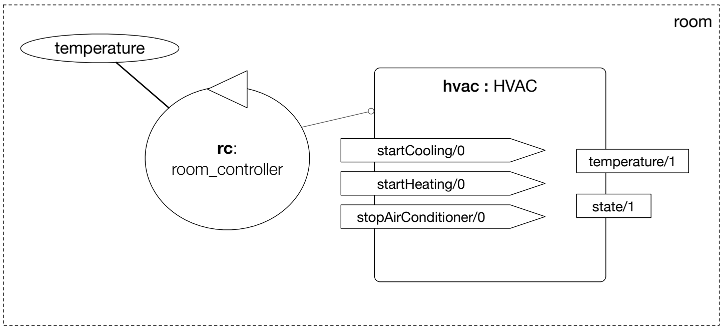


Figure 6.1
The `room_controller` agent acting on and perceiving the `hvac` artifact in the `room` workspace.

– an `hvac` artifact, with the basic actions and observable properties, providing to the agent the means to control the HVAC device, encapsulating and hiding related machinery; and
– a `room` workspace, representing the room location where the agent and the artifact are (logically) situated.

In this simple case, the `hvac` artifact triggers an activity of heating or cooling, that at some point will need to be stopped or started (see the statechart diagram in figure 6.2). It has the following usage interface:

defining the artifact usage interface

- `startCooling` operation to start cooling, switching on the cooling process available in the HVAC device;
- `startHeating` operation to start heating; and
- `stopAirConditioner` operation to stop cooling or heating.

The operations have no parameters.
The usage interface includes also the observable properties that make it possible for the agent to perceive the state of the world (of interest in this example):

- the `temperature` observable property, representing the current temperature of the room, which is meant to be measured by a thermometer component of the HVAC device; and
- the `state` observable property, representing the current state of the HVAC, which could be `idle`, `cooling`, or `heating`.

Separating the programming of the tools and resources in the artifacts from the control and decision making in the agents enables the independent refinement of each of them as long as the interfaces between agents and environment are kept stable, that is, the set of actions and observable properties defining the usage interface

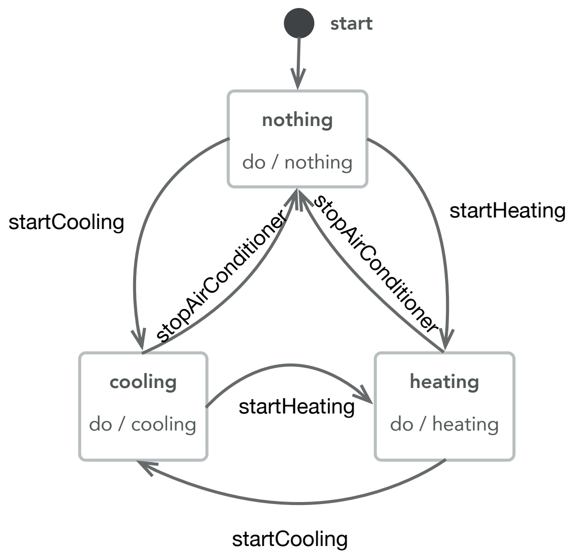


Figure 6.2
Statechart diagram of the `hvac` artifact.

of the artifact. For instance, we can envision reusing the same `hvac` artifact under different control strategies programmed in the `room_controller` agent. The same `room_controller` agent may be also used with different kinds of HVAC physical devices, which could be represented by different artifacts featuring the same interface. The availability of both agent and artifact abstractions promotes thus the *separation of concerns* and allows for *modularity*, which in turn helps abstraction, reusability, and extensibility.

Good Practice for Designing Agents and Artifacts

At the design level, should we start first identifying agents or artifacts? Actually there are no strict rules. Depending on the application, both activities can be done at the same time in an iterative design, or in sequence, identifying first the agents and then the artifacts, that is, defining operations and observable properties to satisfy the actions and beliefs requirements, or inversely, considering beliefs and actions on basis of the observable properties and operations. The most important is to provide the right abstractions for defining an artifact usage interface so that agents can act on them and perceive the right properties of their state.

Having a separate design and programming of the tools and resources in the artifacts from the control and decision making in the agents enables the independent refinement of each of them as long as the set of actions and observable properties defining the usage interface of the artifact is kept stable. The ultimate goal is to be able to reuse the same artifact under different control strategies programmed in agents and to use the same agent on different artifacts that have the same usage interface but are implemented in different ways.

Programming the room controller agent From the decisions and design sketched in the previous section, we program the `room_controller` agent by first defining its goals and then the plans to achieve them.

defining agent goals

We start with programming the agent's *achievement goal*, which represents the task we expect the agent to do. As explained in chapter 4, achievement goals are denoted by the operator `!` followed by a literal. A *literal* is an atomic formula as in logic programming, possibly negated. The achievement goal in our running example can be written as

```
!temperature(TargetValue)
```

The logical *variable* `TargetValue` stands for a concrete number, as in `!temperature(24)`, for instance. The meaning is that the agent has the goal to achieve a state of the world in which the specified room temperature has been reached. This representation explicitly refers to the literal `temperature(Value)`, that is, the same literal used to represent the agent belief about the current temperature. This belief corresponds to the `temperature` observable property perceived from the `hvac` artifact. This property has been designed to represent the current temperature of the environment. This belief is automatically generated and kept updated within the agent's belief base as soon as the agent starts observing the `hvac` artifact. Each time the artifact changes its observable property, the agent automatically perceives the change through a percept, the corresponding belief is updated, and a belief change internal event is generated.

The agent wants to achieve a state of the world in which that property has the specified target value. The goal could be assigned to the agent either dynamically at runtime (e.g., through communication, as described in the next chapter) or when the agent is launched, or it could even be statically specified in the agent source code.

defining agent plans

Given a goal to achieve, we have to program *how* to handle this kind of goal, that is, which *plans* can be used for that purpose. The strategy to define those plans depends on the set of available actions that the agent can use to affect the environment. Such a set is constrained by the physical or merely external environment where the system is situated (for instance, the functionalities provided by a physical HVAC system). Artifacts make it possible to design the logical environment made accessible to agents by defining the set of actions and observable properties—choosing the best level of abstraction from the agent point of view.

Given the set of actions in the usage interface of the `hvac` artifact, a first simple strategy for the plan to achieve the goal `!temperature(TargetValue)` is

1. to start cooling if the current temperature is higher than the target one, and go on cooling until the target temperature is reached;
2. analogously, to start heating if the current temperature is lower; and
3. to stop working when the target temperature is reached.

This strategy can be implemented by the following three plans:

```
@start_cooling // start_cooling is the name for the plan
+!temperature(T) : temperature(C) & C > T
<- startCooling;
   !cool_until(T).

@start_heating
+!temperature(T) : temperature(C) & C < T
<- startHeating;
   !heat_until(T).

@stop
+!temperature(T) : temperature(T)
<- stopAirConditioner;
   println("Temperature reached ",T).
```

In our example, the three plans are triggered by the same triggering event, namely, `+!temperature(T)`, that is, the new goal of achieving the specified temperature.

As shown, an agent can have multiple plans that are *relevant* for the same goal to be achieved, as alternatives to be used depending on the situation. In our example, the three plans have to be used when the perceived temperature is lower than, greater than, or equal to, respectively, the target temperature. The first plan is applicable when the agent believes that the current temperature is higher than the temperature to be achieved. In that case, the action is to start cooling the environment (action `startCooling`) and then go on until the temperature perceived is the target temperature, which is the reason for the subgoal `!cool_until(T)`. The second plan is analogous; it is selected when the temperature is lower than desired and acts by starting the heating system. The third plan completes the strategy: it is selected when the perceived temperature is the desired one and simply stops the air conditioning system (action stopAirConditioner) and prints a message on the standard output.

This first example allows us to introduce and discuss a main feature of agent programming, which is the possibility to decompose a goal into subgoals and then define corresponding plans for each goal. This is an important feature for modularizing the agent design and implementation. In the code of the plans @start_cooling and @start_heating, !heat_until and !cool_until are examples of subgoal creation. When, for example, !heat_until is executed, a new goal event is generated and the execution of @start_heating plan is suspended until the goal has been achieved or a goal failure has been generated. Because it is a subgoal of a goal for which there is an intention in execution, when an applicable plan is found to handle it, no new intentions are created, but rather the plan is stacked on top of the plans already in execution that are related to that same intention.

decomposing goals into subgoals

A possible implementation of the plans for achieving the subgoal !heat_until is as follows:

```
@heat_until_stop
+!heat_until(T): temperature(T)
  <- stopAirConditioner;
     println("Temperature reached ",T).

@heat_until_heat
+!heat_until(T): temperature(C) & C < T
  <- .wait({+temperature(_)}); // wait until temperature changed
     !heat_until(T).

@heat_until_loop
+!heat_until(T): temperature(C) & C > T
  <- !temperature(T).
```

That is, in the case that the target temperature is achieved, then stop the HVAC (plan @heat_until_stop). Otherwise, go on heating—that is, carry on trying to achieve the !heat_until(T) goal—if the temperature is still lower than it should be (plans @heat_until_heat and @heat_until_loop).

In order to avoid repetition in the formula expression in the plan contexts, we can define *prolog-like rules*, refactoring the code as follows:

factorizing the code

```
is_colder_than(C,T) :- temperature(C) &  C < T.
is_warmer_than(C,T) :- temperature(C) &  C > T.
in_range(C,T) :-
    not is_colder_than(C,T) & not is_warmer_than(C,T).

@start_cooling
+!temperature(T) : temperature(C) & is_warmer_than(C,T)    <- ...
```

```
@start_heating
+!temperature(T) : temperature(C) & is_colder_than(C,T)   <- ...

@stop
+!temperature(T) : temperature(C) & in_range(C,T) <- ...

@heat_until_stop
+!heat_until(T)  : temperature(C) & in_range(C,T) <- ...

@heat_until_heat
+!heat_until(T)  : temperature(C) & is_colder_than(C,T)   <- ...

@heat_until_loop
+!heat_until(T)  : temperature(C) & is_warmer_than(C,T)   <- ...
...
```

Finally, we further refine the solution by taking into account a *threshold* to avoid a hysteresis phenomenon. To that purpose, we add a belief `threshold(Th)` to the agent program to keep track of the threshold and revise the predicates as follows:

defining agent beliefs

```
threshold(5).
...
is_colder_than(C,T) :-
    temperature(C) & threshold(DT) & (T - C) > DT.
is_warmer_than(C,T) :-
    temperature(C) & threshold(DT) & (C - T) > DT.
```

So, as well as being used to represent the information about the environment available to the agent, beliefs can be used on the agent side to keep track of any kind of internal information (that is, mental notes), which can be updated by means of special internal actions used in plans. When a belief appears in agent code, we call it an *initial belief*. We subsequently show that beliefs can also be used to represent information received from other agents through communication.

Theoretical vs. Practical Reasoning

In programming agents, one can use two types of reasoning: theoretical reasoning, that is, reasoning over beliefs by querying the belief base and deriving new beliefs; and practical reasoning, that is, reasoning directed toward actions by decomposing goals into subgoals until executing actions. Whereas practical reasoning is captured by the agent plans, theoretical reasoning is captured by *prolog-like rules*. Such rules are used to derive new beliefs from perceived, communicated, or generated ones. They are also used to simplify the writing of conditions used in plans by making them more succinct and easier to read. Using such rules helps to make plan contexts much more compact. They can be used to factorize some of these conditions that appear in the context part of agent plans.

Programming the HVAC artifact Now we consider the implementation of the `hvac` artifact, implementing the usage interface designed in the previous section (shown in figure 6.1). In a real-world application, an artifact implementing an HVAC would wrap the code used to access the physical device, managing the actuators and the sensors. In this example we abstract away from that level, implementing a simulated HVAC instead.

As presented in chapter 5, an artifact is defined by extending the `Artifact` class. Operations are implemented as methods annotated with `@OPERATION`, and observable properties are defined by means of the `defineObsProperty` primitive. They can be accessed using the `getObsProperty` method. From the design given in the previous section, the `HVAC` artifact template has a couple of observable properties—`state` and `temperature`—and operations to start and stop cooling/heating. They are programmed as follows:

```
public class HVAC extends Artifact {

  void init(double initialTemperature){
    defineObsProperty("state","idle");
    defineObsProperty("temperature",initialTemperature);
  }

  @OPERATION void startHeating(){
    getObsProperty("state").updateValue("heating");
    execInternalOp("updateTemperatureProc",0.5);
  }

  @OPERATION void startCooling(){
    getObsProperty("state").updateValue("cooling");
    execInternalOp("updateTemperatureProc",-0.5);
  }

  @OPERATION void stopAirConditioner(){
    getObsProperty("state").updateValue("idle");
  }

  @INTERNAL_OPERATION void updateTemperatureProc(double step){
    ObsProperty temp = getObsProperty("temperature");
    ObsProperty state = getObsProperty("state");
    while (!state.stringValue().equals("idle")){
      temp.updateValue(temp.doubleValue() + step);
      log("Temperature: "+temp.doubleValue());
      await_time(100);
    }
  }
}
```

The name used to define an operation (e.g., `startHeating`) or an observable property (e.g., `temperature`) must then be used in the same way on the agent side to refer to the corresponding action and belief.

In this simulated version, the heating/cooling operations directly change the value of the temperature, increasing or decreasing it, respectively, step by step. This increasing/decreasing process is implemented by the internal operation `updateTemperatureProc`, triggered by the `execInternalOp` primitive.

Defining and running the smart room JaCaMo program To complete our first JaCaMo program, we need to define the main application file specifying the initial set of agents and artifacts that must be created, their location in workspaces, and other configuration details.

JaCaMo application file

```
1  mas smart_room {
2
3    agent rc : room_controller.asl {
4      goals: temperature(21)
5      focus: room.hvac
6    }
7
8    workspace room {
9      artifact hvac: devices.HVAC(15)
10   }
11 }
```

In this case we have a single `room` workspace (line 8), hosting an agent called `room_controller` (line 5; by focusing on the `hvac` artifact situated in `room` workspace the agent joins this workspace), and an artifact called `hvac` (line 9), whose structure and behavior is defined by the Java class whose full name is `devices.HVAC`—that is, the class `HVAC` is stored in a package called `devices`. The application file allows us to instantiate artifacts by specifying parameters (e.g., `15` in the example) that are passed to the artifact constructor `init`.

The application file also allows the specification (line 4) of the initial set of beliefs and goals of the agent (e.g, the goal `temperature(21)` in our case), as well as the specification of the artifacts that the agent wants to observe (i.e., `hvac`) as soon as it is spawned (used in the application file of the instruction `focus` on `hvac` in

```
CArtAgO Http Server running on http://192.168.125.30:3273
Jason Http Server running on http://192.168.125.30:3272
[Cartago] Workspace room created.
[hvac] Temperature: 15.0
[Cartago] artifact hvac: devices.HVAC(15) at room created.
[room_agent] joinned workspace room
[room_agent] focusing on artifact hvac (at workspace room) using namespace default
[room_agent] It is too cold -> heating...
[room_agent] Temperature perceived: 15
[hvac] startHeating
[hvac] Temperature: 15.5
[room_agent] Temperature perceived: 15.5
[hvac] Temperature: 16.0
[room_agent] Temperature perceived: 16
[hvac] Temperature: 16.5
[room_agent] Temperature perceived: 16.5
[hvac] Temperature: 17.0
[room_agent] Temperature perceived: 17
[hvac] Temperature: 17.5
[room_agent] Temperature perceived: 17.5
[hvac] Temperature: 18.0
[room_agent] Temperature perceived: 18
[hvac] Temperature: 18.5
[room_agent] Temperature perceived: 18.5
[hvac] Temperature: 19.0
[room_agent] Temperature perceived: 19
[hvac] Temperature: 19.5
[room_agent] Temperature perceived: 19.5
[hvac] Temperature: 20.0
[room_agent] Temperature perceived: 20
[hvac] Temperature: 20.5
[room_agent] Temperature perceived: 20.5
[hvac] Temperature: 21.0
[room_agent] Temperature perceived: 21
[hvac] Temperature: 21.5
[hvac] stopAirCond
[room_agent] Temperature perceived: 21.5
[room_agent] Temperature achieved 21
```

Figure 6.3
Smart room execution.

the `room` workspace). In this simple example, all the components of the system are defined at initialization time in the application file. However, as introduced in chapter 5 and to be further discussed, elements of the system (e.g., agent, artifact, workspace, goal, and belief) can be defined at runtime by the agents themselves.

In this first implementation, we have a very simple example of an agent perceiving and acting on the `hvac` artifact. The goal-directed behavior is fixed by the user in the application file with the initial goal `temperature(21)`. The agent is merely executing the plans that are possible in the state of the environment under this goal. In the following, we introduce how agents can create goals and adapt their goal-directed behavior to the evolution of the environment.

6.2 Adding Reactivity to the Smart Room

A main feature of agents is the ability to promptly react to events perceived from the environment while also proactively carrying out behavior to achieve goals by executing appropriate actions. To see this in our running example, we extend the capability of the `room_controller` agent so that along with being capable of bringing the environment temperature to the specified level, we also want the agent *to maintain* the desired temperature. That is, after a particular temperature is reached, the room temperature might change, and in that case the agent should *react* to such a change in the environment so as to bring back the temperature to the desired level.

To make the agent able to react to the evolution of the environment, we exploit a main feature in agent programming: programming *reactive behavior* using the agent plans. These plans are triggered and executed in reaction to changes perceived from the environment, or to be more precise, changes in the *beliefs* about the state of the environment.

programming reactive behavior

We thus extend the set of plans of `room_controller` with the following plan:

```
+temperature(C):
     preferred_temperature(T) & not in_range(C,T) &
     not .desire(temperature(_))
  <- !temperature(T).
```

This plan might be triggered each time the belief about the current room temperature changes (that is, represented by the event `+temperature(C)`). In case the new perceived temperature is out of range with respect to some preferred temperature and the agent does not already have an intention about achieving the target temperature (use of the internal operation `.desire`), a new goal `!temperature(T)` is created.

Programming Reactive and Proactive Behavior

Two main kinds of behavior can be programmed in an agent, based on the following simple patterns of code:

- goal-directed behavior, in which a plan is triggered by the creation of some goal (e.g. `!g1`) that the plan aims at achieving
  `+!g1 : ... <- ...`
- belief-directed behavior, in which a plan is triggered by the creation of some belief (in the following, `b`):
  `+b : ... <- ...`

On the basis of these two main triggering patterns, one can program in an agent:

- a reactive behavior in which the agent executes *actions* in reaction to some event, that is, the body of the plan triggered by the creation of the corresponding belief is composed of actions to execute; and
- a proactive behavior in which the agent creates *goals* in reaction to some event, that is, chaining the achievement of goals and subgoals.

  ```
  +b : ... <- !g1.
  +!g1 : ... <- a1; !g2; a2.
  +!g2 : ... <- ...
  ```

updating beliefs

To keep track of the preferred desired temperature, we introduce the new belief `preferred_temperature/1`. The belief can be changed (created or updated) in plans that handle `!temperature(T)` goals. The plans of the `room_controller` agent can be revised as follows:

```
+!temperature(T)
  <- -+preferred_temperature(T);
     !achieve_temperature(T).

+!achieve_temperature(T): temperature(C) & in_range(C,T)
  <- println("Temperature reached ",T).

+!achieve_temperature(T): temperature(C) & is_colder_than(C,T)
  <- println("It is too cold, heating up...");
     startHeating;
     !heat_until(T).

+!achieve_temperature(T): temperature(C) & is_warmer_than(C,T)
  <- println("It is too hot, cooling down...");
     startCooling;
     !cool_until(T).
```

In the first plan, the operator `-+` updates the belief with the new value, by atomically doing a removal of the respective belief (`-preferred_temperature(_)`) and then adding it to the new value (`+preferred_temperature(T)`).

On the artifact side, the HVAC artifact is extended to include a GUI that allows the user to dynamically change the temperature by means of a slider component, to simulate a real environment. As soon as we have changed the temperature of the room (by means of the GUI), the agent reacts accordingly to drive the current temperature to the preferred one.

Interacting with the User by Means of Artifacts

Artifacts can be used in JaCaMo to develop GUI components that allow agents to interact with users. This can be done by developing the GUI elements separately and then exploiting the CArtAgO API to safely access artifacts from these elements, in order to update observable properties, for example. To illustrate this approach, the HVAC artifact is equipped with a `TemperatureSensorPanel` frame—implemented using the Java Swing API—that allows the user to change dynamically the temperature by means of a `JSlider` component.

```
public class HVAC extends Artifact {
  private TemperatureSensorPanel sensorPanel;

  void init(int temp, int prefTemp){
    ...
    sensorPanel = new TemperatureSensorPanel(this,temp);
    sensorPanel.setVisible(true);
  }
  ...
  void notifyNewTemperature(double value){
    getObsProperty("temperature").updateValue(value);
  }
}

class TemperatureSensorPanel extends JFrame {
  private JTextField tempValue;
  private JSlider temp;

  public TemperatureSensorPanel(HVAC hvac, int startTemp){
    setTitle("..:: Temperature Sensor ::..");
    ...
    temp = new JSlider(JSlider.HORIZONTAL, 5, 45, startTemp);
    temp.addChangeListener((ev) ->  {
      JSlider source = (JSlider) ev.getSource();
      int value = (int) source.getValue();
      tempValue.setText(""+value);
      if (!source.getValueIsAdjusting()) {
        hvac.beginExtSession();
        hvac.notifyNewTemperature(value);
        hvac.endExtSession();
      }
    });
    ...
  }
}
```

In the source code of the panel, a listener attached to the slider component is executed each time the slider position is changed. The control flow executing the listener is the one from the Swing toolkit (that is, the

Swing Event Dispatcher Thread). In order to allow an external control flow to interact with artifacts without interfering with the ones used by the environment runtime, an *external session* must be explicitly requested by means of the `beginExtSession` method provided by artifacts. After beginning the session, the external control flow can safely execute methods of the artifact (e.g., `hvac.notifyNewTemperature`) in a mutually exclusive way, without creating interferences. The session must be closed with a `endExternalSession`, specifying whether it was successful or not, so as to release the exclusive access to the artifact.

6.3 Adding Fault Tolerance to the Smart Room

In the first two sections, we explained how to program an agent that is both proactive and reactive. Reactivity also implies the ability to react to failures that can occur when the agent executes actions in open, dynamic, and unpredictable environments. However, at this stage the application is not able to react and handle such kinds of environments. In this section, we investigate how to program an MAS that is able to handle faults that could appear during its execution. Modeling and handling exceptions and failures is an important feature in multi-agent programming, especially in dealing with open, dynamic, and unpredictable environments. In JaCaMo, this involves both the agent and the environment dimensions:

- an operation on the artifact side can be programmed so as to interrupt its execution, generating a *failure action event* and reporting information about the problem that has occurred; and
- on the agent side, the plan executing an action (operation) that fails is interrupted and a *goal deletion* event (event preceded by `-!`) is generated, so that it could be properly managed by plans designed for that purpose, called *contingency plans*.

Handling failures in artifacts We want to model in our example the situation in which the HVAC becomes broken, so that operations like `startHeating` (or `startCooling`) will not work. Because the problem has to be handled at the agent level and not internally in the artifact via the use of classic exception mechanisms, it is implemented with a *failure action event* generated with the primitive `failed` as follows:

defining failure action events

```
public class HVAC extends Artifact {
    ...
   @OPERATION void startHeating(){
     if (<check for the presence of failures>){
       failed("HVAC Broken Failure","broken_hvac",FAILURE_CODE)
     } else {
       getObsProperty("state").updateValue("heating");
       execInternalOp("updateTemperatureProc",0.5);
     }
   }
   ...
 }
```

The primitive `failed` is used to interrupt operation execution. The parameters of this primitive are used to share the context of execution raising the failure with the agent that executed the action. The parameters specify a failure description textual message (`HVAC Broken Failure` in the example above) and, optionally, a structure suitable to be processed on the agent side (`broken_hvac(FAILURE_CODE)`) that represents the reason for the failure.

Handling failure in the agents On the agent side, we can write down one or more contingency plans reacting to the failure notified by the goal deletion event. The reaction to the failure is carried out by inspecting the failure reason through the `env_failure_reason` that annotates the generated goal deletion event:

defining contingency plans

```
+!temperature(T) : temperature(C) & C < T
  <- startHeating;
     !heat_until(T).

-!temperature(T) [error(broken_hvac(CODE)),error_msg(Msg)]
  <- println(Msg);
     !inform_owner(broken_hvac(CODE)).
```

Different kinds of contingency plans can be specified to react to different kinds of failure defined as parameters to the `failed` method in the artifact; the parameters are reported by the annotation attached to the generated event.

6.4 Making the Smart Room Adaptive

As the agents of the smart room are currently programmed, they are not prepared to adapt their behavior to the evolution of the environment (e.g., change of user preferences, deletion, and introduction of some artifact) or of the agents themselves (e.g., addition of new plans). However, an important feature for autonomous agents is *adaptivity*, that is, the ability to flexibly and dynamically adapt their behavior. Agent programming languages provide first-class support for managing intentions and goals in agent programs.

Let us consider in our example a further extension to allow the introduction of dynamics in the environment. For instance, by means of a control panel, we will let users change dynamically the preferred temperature of the room. In that case, the room_controller agent has to react not only to changes to the room temperature but also to changes to the preferred temperature, which directly determines the goal of the agent. In particular, if a new preferred temperature is specified while the agent has an ongoing intention to achieve a previous preferred temperature, then the agent has to drop that intention and create a new one. To that purpose, the belief preferred_temperature is no longer created by the agent but is meant to be an observable property of the hvac artifact in the environment. The value of this observable property can be changed dynamically by the user.

managing goals

To handle this new feature, a first implementation of the new plans in the agent program is

```
@p1
+preferred_temperature(T):
     temperature(C) &
     not in_range(C,T) & not .desire(temperature(_))
  <- println("Reacting to temperature preference change");
     !temperature(T).

@p2
+preferred_temperature(T):
     temperature(C) &
     not in_range(C,T) & .desire(temperature(T1))  & T1 \== T
  <- println("Reacting to temperature preference change");
     .drop_desire(temperature(T1));
     stopAirConditioner;
     !temperature(T).
```

The first plan @p1 is chosen when the agent does not yet have the intention[1] to achieve a target temperature; in that case, the new goal is simply created. The second plan @p2 is used instead when the agent is already pursuing the goal of achieving some temperature and the new preferred value is different; in that case, the agent must give up the current intention, using the internal action .drop_desire before creating the new goal.

managing intentions

This solution has a problem if the agent concurrently perceives that there has been a change in the temperature and the preferred temperature. For example, what happens if a change to temperature(T) is perceived so that a new intention executing plan p1 is instantiated, but a change to preferred_temperature(T) occurs before the subgoal !temperature(T) is created? In that case, plan p2 would be chosen for execution because it is true

1. Recall that an intention is a desire (goal) that the agent is committed to achieve by means of a particular plan. Although we could have used .drop_intention, the .drop_desire action is more general, and therefore we used it in the example although the text refers to intention.

that `not .desire(temperature(_))`. So two interfering intentions would be in execution. This problem is an example of *interferences* that can occur when the agent interleaves checks and action blocks that should be executed *atomically*; the *atomic plan* feature can be used to avoid such a problem. In this case, this concerns checking the condition `.desire(temperature(_))` in the context and the action `!temperature(T)` in the plan body. Such an atomic behavior can be enforced by annotating plans `@p1, @p2, @p3` with the `atomic` attribute as follows:

defining atomic plans

```
@p1[atomic]
+temperature(C):
     preferred_temperature(T) & not in_range(C,T) &
     not .desire(temperature(_))
  <- // ...
     !!temperature(T).

@p2[atomic]
+preferred_temperature(T):
     temperature(C) &
     not in_range(C,T) & not .desire(temperature(_))
  <- // ...
     !!temperature(T).

@p3[atomic]
+preferred_temperature(T):
     temperature(C) &
     not in_range(C,T) & .desire(temperature(T1)) & T1 \== T
  <- // ...
     !!temperature(T).
```

Annotating the plan label (e.g., `@p1`) with an `atomic` tag ensures that no other intention will be selected to interleave execution with this plan.

We can notice that the plan bodies have been updated, replacing the issuing of a subgoal `!temperature` with `!!temperature`, which accounts for issuing a new goal *to be managed with a separate intention*. That is, `temperature` is no longer a subgoal of the current intention but is a separate, *parallel goal*. In this specific case, this feature is needed because if we instantiate a subgoal from an atomic plan, the atomic constraint is kept also when executing the plan triggered for handling the subgoal: it is an attribute at the *intention* level. This would not be the desired behavior in our case, because while achieving the new temperature the agent should be able to react and interleave different intentions, while the atomic plan itself can finish soon.

defining parallel goals

Adapting the Behavior by Dynamically Changing the Plan Library

As discussed in this chapter, agents can control their intentions and, as discussed previously, they can control their beliefs and execution as well. For instance, agents can create, inspect, and drop their own intentions, revising thus the goals to which they have committed. Nonetheless, agents can also control their plans: the plan library can be managed as well as other mental components. To be handled in a program, plans have to be represented as terms, and for that we enclose them with `{` and `}`. For instance, `{+!g <- .print(hello)}` can be used as a term for some literal or internal action. The main operations to handle the plan library are briefly presented in the sequel.

- **Inspection** The internal action `.relevant_plans(TE,L,LL)` can be used to retrieve in list `L` all relevant plans for event `TE`; the `LL` argument is optional and correspond to the labels of plans in `L`. For example, `.relevant_plans({+temperature(_)},L)` unifies `L` with a list of plans relevant for the event `+temperature(_)`.
- **Addition** The internal action `.add_plan(P)` is used to add the plan term `P` in the plan library. For example, `.add_plan({@l1 +!g : today(ok) <- .print(hello)})` adds the plan `@l1` into the plan library.
- **Deletion** Plans are removed by their labels using the `.remove_plan(L)` internal action. For example, `.remove_plan(l1)` removes the plan identified by label `l1` from the plan library.

Agents not only can control their own mental state (beliefs, intentions, and plans), but they can influence others! Whereas communication performative verbs like `tell` and `achieve` are used to influence others' beliefs and intentions, the performative verb `tellHow` influences others' plan library by adding a plan in the receiver plan library. Of course, an agent may refuse to accept beliefs, intentions, and plans sent by others. The next chapter introduces communication among agents in details and explains these issues.

These programming features improve autonomy (agents control their mental state and program) and sociability (agents can influence others' mental state and program). The following meta plan illustrates these features:

```
-!G[error(no_relevant)] : teacher(T)
    <- .send(T, askHow, { +!G }, Plans);
       .add_plan(Plans);
       !G.
```

This plan reacts to a goal failure because there is no relevant plan, and because `G` is a variable, it is used for any goal. The plan consists of asking a teacher `T` for plans to achieve goal `G`, adding the answer (a list of plans unified with `Plans`) into the plan library and retrying the failed goal.

6.5 What We Have Learned

In this chapter, we learned how to program a first simple multi-agent program in which one agent perceives and acts in a dynamic environment composed of a simple artifact. The key concepts and competences presented are

- Designing and programming an *agent* with proactive behavior (it acts in order to achieve a goal according to some plans), and reactive behavior (it promptly reacts to relevant changes in the environment). The programming tools we have for this design are
  - agent goals and beliefs;
  - goal-directed and belief-directed plans (used to achieve goals and to react to environment changes);
  - contingency plans (used to handle failures of action execution in open, dynamic, and unpredictable environments); and
  - the manipulation of the agent internal mental state (used to adapt the agent behavior to the evolution of the environment).
- Designing and programming an *environment* in which agents are situated. The programming tools we have for this design are
  - artifact usage interface (in terms of operations and observable properties);
  - environment processes and functionalities (in terms of artifact behavior); and
  - wrapping existing nonagent resources (such as objects, legacy code, and libraries) as artifacts.

Pursuing a step further in the definition of this simple MAS, this chapter also presented how to handle failure in the artifact and in the agent level as well as to adapt the behavior of the agent in a dynamic environment.

Exercises

Exercise 6.1 In exercise 4.1 on page 50, you had to write a plan for having a cake by buying it at a pastry shop. Create a plan pattern for an agent that is blindly committed (i.e., it will keep trying to buy the cake until that is achieved). Change the pattern so that the agent has a single-minded commitment toward that goal instead; that is, it will no longer try to achieve that goal if it believes that it has become impossible to buy the cake (e.g., all shops are closed).

Exercise 6.2 In exercise 4.2 on page 50, you had to write a plan for baking a cake. Write a contingency plan to be used in case the plan for achieving the goal fails (e.g., you do not have the recipe for that type of cake). In that case, you go and buy a cake of that type at the pastry shop.

Exercise 6.3 The smart room goes green, and a new green HVAC has been purchased. Design and implement a `GreenHVAC` artifact as an extension of the basic HVAC, including the following features:

- The energy consumed is now observable in KWh since it was switched on, assuming that
  - the power P of the device in kW is passed as a parameter of the artifact; and
  - the energy consumed can be computed as $P * DT$, where DT is time elapsed since it was switched on (suggestion: internal operations can be used to this purpose, together with `await_time`).
- It provides a `setGreenModeOn` operation that makes it possible to reduce by half the power used and an observable property representing whether the green mode is either on or off.

Then, extend the functionality of the `RoomControllerAgent` in order to apply a green policy, so that

- it activates the green mode if the HVAC has consumed more than $Threshold_{energy}$ energy in the last $Threshold_{period}$ period ($Threshold_{period}$ could be expressed in seconds); and
- it switches the device off if the HVAC has consumed more than $Threshold_{energy}$ energy in total.

Exercise 6.4 A domestic robot has the goal of serving beer to its owner. Its mission is quite simple: it just receives some beer requests from the owner, goes to the fridge, takes out a bottle of beer, and brings it back to the owner. However, the robot should also be concerned with the beer stock (and eventually order more beer using the supermarket's home delivery service) and some rules hardwired into the robot by the Department of Health (in this example, this rule defines the limit of daily beer consumption). The system is composed of three agents: the robot, the owner, and the supermarket.

7 Programming Multiple Agents Interacting in an Environment

Agent interactions are essential aspects of multi-agent systems. The basic mechanisms introduced in chapter 4 on the agent dimension and in chapter 5 on the environment dimension enable the definition of various sophisticated types of interactions. In this chapter, we go deeper into these aspects, focusing on both agent direct interaction based on speech-act–based communication and agent indirect interaction mediated through environment artifacts. Seeing them from a practical point of view in this chapter, we consider an extension of the smart room scenario of the previous chapter that now involves multiple agents.

7.1 Programming a Smart Room with Multiple Agents

We start by considering an extension of the smart room scenario in which direct communication among agents is adopted and an agent is in charge of the coordination of the agents. The extended scenario accounts now for allowing people who enter the room to set up their preferred temperature. The `HVAC` artifact cannot be initialized now with a single preferred temperature.

The idea is that each user is supported by a kind of *personal assistant* agent that acts on his/her behalf in interacting with the other agents and artifacts in the application. The `personal_assistant` agent is thus defined to manage and generate its user's preferences on temperatures given the activities that she/he may undertake in the room. Activities may consist of, for example, reading a book, watching a movie, or practicing some sport. Because each user may have his/her own way of associating some preferred temperature to an activity, depending on the context, encapsulating such knowledge within a personal agent is a good approach.

Interacting by agent communication As introduced in chapter 4, the interaction between `personal_assistant` agents and the `room_controller` agent is realized by the communicative action `.send`. The first argument of this action corresponds to the name of the agent that should receive the message (agent `rc` in our *defining communicative actions*

subsequent example). The second argument corresponds to the *performative verb* denoting the intention of the sender and helps the receiver to interpret the third argument that contains the actual content of the message. In our example, the sender intends the receiver to interpret the content as a belief, and thus the performative verb used is tell. The *content* corresponds to the preference on the temperature value for the user that the agent sending the message represents: `pref_temp(T)`. For instance, when the `personal_assistant` agent `pa1` executes

```
.send(rc, tell, pref_temp(10))
```

the receiver agent, `rc`, will have the belief `pref_temp(10)[source(pa1)]` automatically included in its belief base; no programming is necessary on the receiver code for that.

By sending messages, a JaCaMo agent is thus changing the mental state and influencing the behavior of the receiver, whether the content of messages are beliefs, goals, or plans. For instance, messages from `personal_assistant` agents may turn into belief additions in the `room_controller` agent's belief base; corresponding events are produced, and the `room_controller` agent can then react to them with the following plan:

reacting to received messages

```
+pref_temp(UT)[source(Ag)]
   <- .println("New preference from ", Ag, " = ",UT);
      // if previous preference of Ag is different
      if (pref_temp(Y)[source(Ag)] & UT \== Y) {
         // remove the previous preference
         -pref_temp(Y)[source(Ag)];
      }
      ...
```

In the previous plan, annotations added to the produced belief such as `source(...)` are used so that the receiver can identify where the elements of its own mental state originated. In the case of the message sent above, the value of variable `Ag` is `pa1`.

Speech Acts

Communication languages for autonomous agents have been strongly influenced by speech act theory, in particular the work of the philosophers of language Austin (1962) and Searle (1969). In practice, this implies messages exchanged by agents, having a clear separation of the actual content from the intent of the sender, which is also explicitly represented and expressed as performative verbs. For example, an agent may send a message to change another agent's beliefs or another agent's goals. On the more practical side, the first agent communication language was KQML (Mayfield et al. 1996), and a lot of work followed on the FIPA agent communication language which was based on the work initially reported by Bretier and Sadek (1996). For further classic references and all the basics on agent communication, see chapter 7 in Wooldridge (2009).

Performative Verbs

JaCaMo implements a set of performative verbs. For further details on advanced features, such as filtering reception of beliefs and redefining the semantics of performative verbs, see Bordini et al. (2007), 118. The performative verbs are the following:

- **`tell`** The sender expects that the receiver will have the content of the message included in its belief base annotated with the sender as the source of that information. For example, the action `.send(a,tell,v(10))` done by agent `b` is expected to result in the inclusion of the belief `v(10)[source(b)]` in the belief base of agent `a`.
- **`untell`** The sender expects that the receiver retracts from its belief base the content of a previous message. The content corresponds to the belief the sender no longer holds. For example, the action `.send(a, untell, v(10))` done by agent `b` will retract the belief `v(10)[source(b)]` from the belief base of `a`.
- **`achieve`** The receiver is expected to have the content of the message as a new goal with the source also annotated. For example, the action `.send(a,achieve,g(10))` done by agent `b` will include the goal `!g(10)[source(b)]` for the agent `a`.
- **`unachieve`** The sender expects the receiver to drop the goal of achieving a state of affairs corresponding to the message content. For example, the action `.send(a,unachieve,g(10))` done by agent `b` will drop the goal `!g(10)` for the agent `a`.
- **`askOne`** The sender wants to know whether or not the receiver believes in the content of the message (any belief that matches the pattern in the message content). If the receiver does so, it answers with one of its own belief; otherwise it answers **`false`**. For example, if `b` executes `.send(a,askOne,v(X))`, `a` will, normally, automatically execute `.send(b,tell,v(10))` in case it believes `v(10)`. This performative can be used synchronously when a fourth argument is used. For example, if

one of agent `b`'s intentions executes `.send(a,askOne,v(X),A)`, this intention is suspended while it waits for the answer. When the answer from `a` arrives, it is unified with the fourth argument and the intention resumed so that the next line of code in the plan body can already make use of the received response.

- **`askAll`** This performative is similar to `askOne` but it retrieves all the beliefs instead of one.
- **`askHow`** The sender asks the receiver for plans that can be used to handle some particular event. For example, an agent sending `.send(a,askHow,{+!g(_)},P)` will have in the variable `P` a list of plans from agent `a` that can be used to achieve goal `g`, and it can then add these news plans into its own plan library, if needed.
- **`tellHow`** The sender informs the receiver of a plan it has in its plans library. For example, agent `a` receiving `.send(a,tellHow, {+!start[source(X)] <- .print("hello from ",X).})` will add the plan `+!start[source(X)] <- .print("hello from ",X).` in its plan library.
- **`untellHow`** The sender requests that the receiver disregards a certain plan (i.e., deletes that plan from its plan library).

Programming the personal assistant agents The `personal_assistant` agent is a personal assistant aimed at managing the preferences of its user. Because controlling the temperature is already done by the `room_controller` agent, we keep this separation of concerns: we want the decisions about the room temperature to be done collectively by all users, but the control is kept under the responsibility of the `room_controller` agent, whereas the preferences that are different for each user are managed in the `personal_assistant` agent of this user.

The user preferences are stored as beliefs in the `personal_assistant` agent's mental state as follows:

```
preferred(Activity, Value)
```

where `Activity` can be `reading`, `watching`, `cooking`, `sport`, and so forth, and `Value` can be `high`, `medium`, or `low`.

Being different for each user and thus for each `personal_assistant` agent, these preference beliefs are set in the application file (the `.jcm` file; see page 97). In order to compute the preferred temperature, rules are written within the `personal_assistant` agent code as follows:

```
// Computation of the preferred temperature value for the current activity
pref_temp(T) :-
   activity(A) &      // current user activity (from UserGUI)
   preferred(A,L) &  // user preferences (given in the  application file)
   level_temp(L,T).  // the mapping below

// Mapping low, medium, and high temperature levels into numbers
level_temp(low,    10).
level_temp(medium, 20).
level_temp(high,   30).
```

This code represents the beliefs and rules used by the agent to decide on the room temperature to request. We clearly see here the difference between this and the previous chapter in which all the information and decisions were managed by a single agent. The interest in using `personal_assistant` agents to handle such information is that they are personal and that the agent can take appropriate decisions while keeping them private.

To interact with the human user, the `personal_assistant` agent uses a `UserGUI` artifact. This artifact embeds a GUI—based on Java Swing in this case—to enable interaction with the user. The artifact has one observable property, `activity`, describing which activity the user is currently doing (see figure 7.1). The property is properly updated depending on the user action on the GUI.

The actions of the `personal_assistant` agent are determined by two plans. One of these is for the initialization: the agent creates an instance of its own `UserGUI` artifact so that its user can interact with it, and the agent clearly needs to focus on it (in order to perceive any changes made by the user).

```
+!create_GUI
   <- .my_name(Me); // variable Me unifies with the agent's name
      // creates a unique name for the artifact
      .concat(gui,Me,ArtName);
      makeArtifact(ArtName, "gui.UserGUI", [], ArtId);
      focus(ArtId).
```

The name specified for the `UserGUI` artifact in this case is based on the agent name, retrieved by the internal action `.my_name`.

The second plan reacts to changes in the user activity by sending the preferred temperature to the `room_controller` agent. The changes in the activity are perceived by changes in the observable property `activity` from the `UserGUI` artifact on which the agent is focusing.

```
+activity(A) : A \== none
   <- ?pref_temp(T); // retrieve user preferred temperature from BB
      .send(rc,tell,pref_temp(T)). // and sends it to RC agent
```

Note that in this plan we used `?pref_temp(T)`. This is called a *test goal* and, *querying the belief base* different from *achievement goals*, these are used to retrieve information from the agents' belief base within the course of action determined by an executing plan. In this example, if a matching belief can be found in the belief base, variable T will be instantiated with a particular value; otherwise, a plan of the form +?pref_temp(T) : ... <- ... is looked for, and if none is found, the test goal fails.

defining agent types

The code shown above defines an *agent type* (and corresponding .asl file) that can be used to create several agents. All these agents follow the same set of plans, rules, initial beliefs, and goals written in the .asl file. These can be changed and new ones can be added at runtime through communication, perception, or reasoning. Agents of the same type differ only in the initial beliefs and goals set at their initialization and of course in the different acquired beliefs and adopted goals over their own execution histories, depending on the individual interactions with different parts of the environment and other agents.

Revising the room-controller agent When receiving the various preferred temperatures sent by the personal_assistant agents, the room_controller agent has to handle all of them. It generates a new goal !temperature(T) with respect to the temperature to target by acting on the hvac artifact. The generation of this new goal raises a set of questions. Should it just react and change the temperature each time a personal_assistant agent requests it to do so? Should it wait a certain amount of time, compute an average of the incoming preferences, and then act? Which strategy should it use?

There is no unique and definitive answer to these questions. We define a very basic and simple strategy to handle preferred temperatures. We leave as an exercise the development of more sophisticated strategies. Also, the communication here is basic in the sense that there is no sophisticated interaction protocol between the room_controller and personal_assistant agents to handle the decision process between both kinds of agent (e.g., some form of negotiation). In section 7.2, the communication is improved to overcome the centralized decision realized in the room_controller agent.

handling reception of beliefs

The following plan is added to the room_controller agent so that it can react to a received pref_temp from another agent (note the annotation source(Ag) to get the source of the belief, which in this case will be the name of the agent who sent the information).

```
+pref_temp(UT)[source(Ag)]
   <- .println("New preference from ", Ag, " = ",UT);
      // if previous preference of Ag is different
      if (pref_temp(Y)[source(Ag)] & UT \== Y) {
         // remove the previous preference
6           -pref_temp(Y)[source(Ag)];
7        }
8        ?average_pt(T);
9        .drop_desire(temperature(_));
10       .println("Creating a new goal to set temperature to ",T);
11       !temperature(T).
```

When it receives the preferred temperature from some agent (line 1), this plan updates the agent beliefs by retracting from its belief base any previous `pref_temp` information sent by the same agent (lines 3–7), so that there is only one preference for each `personal_assistant` agent in the `room_controller` agent belief base. This updating could be removed if `personal_assistant` agents are programmed to send untell messages retracting previous preferences before sending new preferences to `room_controller` agent. However, even if untell messages are used, it is worth including in the `room_controller` agent a behavior enabling it to update its beliefs. In an open system, this would be good practice as one cannot expect that the agents entering the system care about retracting all previous assertions.

The agent then checks its mental state about the value of the average of the current preferred temperatures (line 8), which is computed as follows:

```
// gets the average of preferred temperatures sent by the
// personal  assistant agents
average_pt(T) :- .findall(UT, pref_temp(UT), LT) &
                 LT \== [] &
                 T = math.average(LT).
```

Finally, before generating the new `!temperature` goal, the agent drops any existing `!temperature` goal under consideration using the special internal action `.drop_desire(temperature(_))`. The `average_pt(T)` rule shown previously also uses two internal actions, `.findall` and `math.average`, the latter of which is actually a function, so it represents a term (the one returned by the function) rather than a predicate or action. Internal actions were discussed in chapter 4.

using internal actions

Deploying and executing As explained in the previous chapter, the application file specifies the initial set of agents, workspaces, and artifacts that have to be created in launching the application. In this case, we populate the `room` workspace with several `personal_assistant` agents (e.g., `pa1`) with some initial beliefs, accompanying the `room_controller` agent (`rc`), focusing on and using the `hvac` artifact situated in the `room` workspace.

```
mas step1 {

    agent pa1 : personal_assistant.asl {
        beliefs: preferred("reading", high)
```

```
                    preferred("watching", high)
                    preferred("cooking", high)
                    preferred("sport", medium)
            join: room
        }

        ...

        agent rc : room_controller.asl {
            focus: room.hvac
        }

        ...
    }
```

The design and implementation adopted so far is an example of a *centralized* coordination schema in which a specific agent—in this case, the `room_controller` agent—is in charge of managing the interaction with all the other agents (the `personal_assistant` agents). However, in some application scenarios, this coordination schema, which is the simplest one, may not be the most effective one, because it could introduce a bottleneck in managing the interactions and a single point of failure. In the following, we show how a more decentralized coordination schema can be defined.

7.2 Decentralizing the Coordination with Interaction Protocols

In MAS, more decentralized coordination schemes could be adopted (see Wooldridge [2009] for an overview). A common strategy is to distribute the responsibility of the coordination among the agents by designing proper *interaction protocols* based on communicative actions.

defining interaction protocols

In our scenario, we consider a revised strategy to decide the temperature to be set in the room. We refer to it as a *fair room* multi-agent system. Instead of adopting a centralized decision process—based on the `room_controller` agent—we want to decentralize the decision, using a *voting* procedure involving the `personal_assistant` agents. To that aim, we define a very simple interaction protocol whereby the `room_controller` agent starts the voting procedure involving the `personal_assistant` agents present in the room workspace (figure 7.1). As in section 7.1, the voting is triggered by changes in the preferred temperatures received by the `room_controller` agent.

The interaction protocol used is a simple sequence of message exchanges involving the `room_controller` agent and the `personal_assistant` agents. The sequence of exchanges starts with the opening of voting by the `room_controller` agent, proposing a closed list of temperature options on

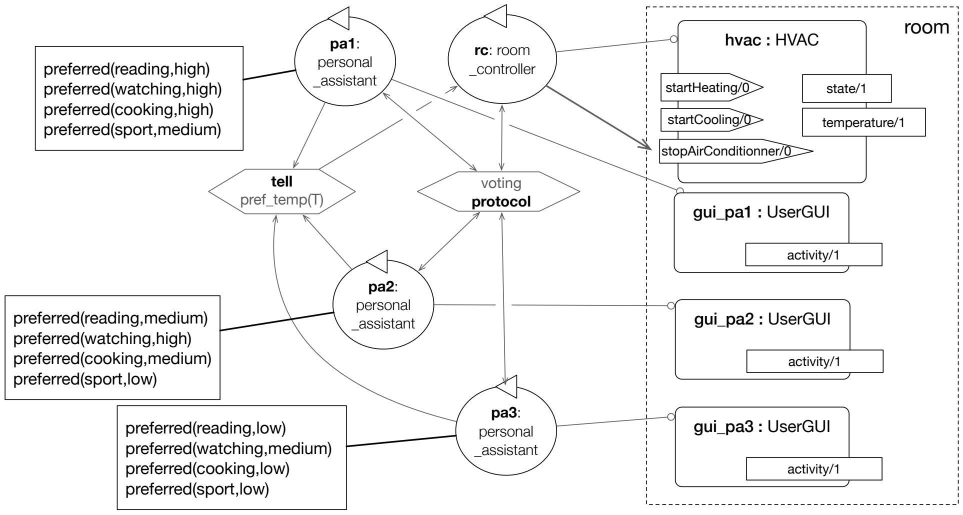


Figure 7.1
Fair room multi-agent system.

which the `personal_assistant` agents are requested to vote by declaring their preference ranking. Once all votes have been received, the `room_controller` agent decides the room temperature using the Borda count method (for an overview of voting procedures used in computational social choice theory, see Wooldridge [2009]. As shown in figure 7.2, the protocol has three moments:

defining broadcast communicative actions

1. **open_voting** The `room_controller` agent first sends a message to all the `personal_assistant` agents present in the system by using the `.broadcast` communicative action with similar arguments to the `.send`, except that the targeted receiver is not specified. This corresponds, for example, to the action:

   ```
   .broadcast(tell, open_voting(p1, [10, 20, 25], 1000))
   ```

   with which the `room_controller` agent informs all the agents of the system that a voting session has been opened. This voting session is identified by `p1`, with temperature options offered `[10, 20, 25]` (a list of preferred temperature candidates for the election), and a statement that the voting will close in 1,000 milliseconds.
2. **ballot** The `personal_assistant` agents answer the `open_voting` message by sending a ballot message composed of the session identification (so that the receiver knows which vote it is about) and its preference. For example:

   ```
       .send(rc, tell, ballot(p1,20))
   ```

3. **close_voting** When all votes have been received or the timeout has expired, the `room_controller` agent announces the closing of the voting with the final result. For example:

```
    .broadcast(tell, close_voting(p1, 20))
```

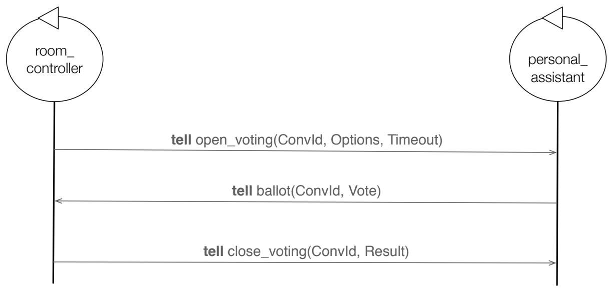

room_
controller
personal_
assistant
tell open_voting(ConvId, Options, Timeout)
tell ballot(ConvId, Vote)
tell close_voting(ConvId, Result)

Figure 7.2
The `voting` interaction protocol.

An interaction protocol is a global description of the expected exchanges of messages issued from the parties involved in the interaction. Even though the specification is global, its implementation is split in the two types of agents we have (`room_controller` agent and `personal_assistant` agents) who are expected to enact this protocol. Note that because in this simple example only one agent receives the votes, there is still a single point of failure. An alternative, although with higher communication complexity, would be for all ballots to be broadcast.

The next sections describe the implementation. The deployment of the application is similar to the one described in the previous section, page 97: the same set of agents with similar preferences are launched in the same workspace in which the same set of artifacts are deployed.

Programming the interaction protocol in the `room_controller` agent The `room_controller` agent initiates the `voting` protocol whenever a new preferred temperature is received from any `personal_assistant` agent. After generating a conversation ID (`!get_id(Id)`), it generates the list of options, denoted as `Options`, on which it would like the `personal_assistant` agents that are present in the room to vote (sending a `.broadcast` message). The conversation ID helps the `room_controller` agent manage several instances of voting procedures in parallel, possibly with the same `personal_assistant` agents. It then waits (cf. internal action `.wait`) until all agents have voted `all_ballots_received(Id)` or the timeout has expired (`4000`), and then it closes the voting `!close_voting(Id)`.

```
+!open_voting
   <- !get_id(Id);
      .findall(T,pref_temp(T)[source(_)],Options);
      .broadcast(tell, open_voting(Id, Options, 4000));
      .wait(all_ballots_received(Id), 4000, _);
      !close_voting(Id).
```

The condition `all_ballots_received(Id)` appearing in the `.wait` action is specified by the following rule that checks that all `personal_assistant` agents expected to participate in the voting process have sent their ballots.

```
  all_ballots_received(Id)
  :- .all_names(L) & .length(L,NP) &        // number of voters =
     .count(ballot(Id,_)[source(_)], NP-1). // number of votes
                                            // (RC doesn't vote)
```

Closing the voting consists of processing the ballots according to the Borda count method (cf. the `!borda_count(Id,Winner)` goal, the plan for it is omitted here but is available in the code accompanying the book), stopping any `!temperature` goals under achievement (cf. internal action `.drop_desire`), and creating a new goal `!temperature(T)` before announcing the closing of the voting process to all agents (cf. the `.broadcast` internal action).

```
+!close_voting(Id)
  <- !borda_count(Id,Winner);
     .println("New goal to set temperature to ",Winner);
     .drop_desire(temperature(_));
     !temperature(Winner);
     .broadcast(tell, close_voting(Id,Winner)).
```

As we can see, two of the three steps of the `voting` protocol are programmed in the `room_controller` agent. The missing step is programmed in the `personal_assistant` agent.

Programming the interaction protocol in the `personal_assistant` agents In participating in the voting protocol by reacting to the creation of an `open_voting` belief, a `personal_assistant` agent sends its vote to the agent that sent the `open_voting` message (use of the annotation `source` in the belief `open_voting`):

```
+open_voting(ConvId, Options, TimeOut)[source(Sender)]
   <- ?pref_temp(Pref);
      ?closest(Pref,Options,Vote); // the individual strategy
      .print("My vote is ",Vote);
      .send(Sender, tell, ballot(ConvId, Vote)).
```

Its vote is computed by ordering the options according to its own strategy (e.g., proximity to their preferred value):

```
closest(X,[H|T],H) :- X > H.
closest(X,[H1,H2|T],H1)
   :- X < H1 & X > H2 & H1-X <= X-H2.
closest(X,[H1,H2|T],H2)
   :- X < H1 & X > H2 & H1-X > X-H2.
closest(X,[H],H).
closest(X,[H|T],V)
   :- closest(X,T,V).

+open_voting(ConvId, Options, TimeOut)[source(Sender)]
   <- ?pref_temp(Pref);
      ?closest(Pref,Options,Vote); // the individual strategy
      .print("My vote is ",Vote);
      .send(Sender, tell, ballot(ConvId, Vote)).

{ include("base-pa.asl") }
```

Performative Verbs in Interaction Protocols

Note how we chose in the preceding code excerpts, to program the interaction protocol by using the performative verb `tell`, making the protocol progress by agents reacting to the new beliefs stated in the messages' content.

We could also have implemented this protocol in a way in which the `room_controller` agent requested the `personal_assistant` agents to act on voting. In that case, the message sent would have the `achieve` performative verb rather than `tell`, for example `.broadcast(achieve,vote(Id,Options,4000))`. In that case, the plan for `+open_voting(...)` should be changed to have `+!vote(...)` as triggering event.

Yet another possibility would be to ask the agent for its vote using a (synchronous) `askOne` message.

Interestingly, the `askHow` performative verb could be used if a new agent in this open multi-agent system did not know how to vote. The `room_controller` could use `tellHow` to send the `personal_assistant` agent a plan for how to vote. For details, see Bordini et al. (2007).

7.3 Environment-Mediated Coordination

Regarding interaction, approaches adopted in multi-agent system take a strong inspiration from the human world. Humans are used to interact using different kinds of approaches and media. Among the possible taxonomies, approaches can be categorized as *direct interaction* and *indirect interaction*.

direct and indirect interaction

- In the *direct interaction* approaches we abstract from the communication *medium*. Speech-based communication is a common example, and another example is given by sign language. In this case, it is not really important whether the communication occurs face-to-face, through a cell phone, or by video streaming: participants, who typically must be in the same temporal frame, exchange messages using a language to encode them.
- In the *indirect interaction* case, the medium enabling the interaction is a first-class entity, decoupling participants and providing functionalities that could be useful to manage the interaction and their coordination. The medium is typically some kind of shared physical object of the environment. A simple example is a blackboard, supporting forms of sharing information. Another example is a street sign suggesting some direction or barriers placed for defining queues. Some media are explicitly devoted to a coordination functionality: a traffic signal, for example, or ticket dispensers for managing service at supermarkets. A well-known example of environment mediated coordination is *stigmergy* (Theraulaz and Bonbeau 1999), which is used in nature by ants as well as by humans and is exploited also in MAS (Van Dyke Parunak 1997; Ricci et al. 2007).

Analogous to the human world, in the MAOP approach we can exploit both direct interaction, realized by means of agent communication languages based on speech acts, and indirect interaction, using artifacts designed as communication and coordination media. As in the human case, these two main categories should not be considered mutually exclusive: conversely, they can be effectively integrated and used in synergy when designing systems. In the following, we exploit indirect interaction in our smart room scenario to improve the collective decision on how the room temperature will be set.

Introducing a voting machine as a coordination artifact In our scenario we design a proper shared *coordination artifact*, encapsulating and regimenting the coordination process ruling the interaction among the agents. The coordination artifact

defining coordination artifacts

encapsulates the voting mechanism described in the previous section. This solution alleviates the code of the agents by transferring part of their computation to shared artifacts, leaving them the decision part. Compared with the centralized version, the decision process is still distributed because of the voting strategy that is still part of each `personal_assistant` and `room_controller` agent.

As in the previous section, from the reception of new preferences from `personal_assistant` agent (see figure 7.3), the `room_controller` agent initiates the voting protocol. For that purpose, it creates a `VotingMachine` artifact with a set of temperature options on which agents can vote, the list of `personal_assistant` agents that are allowed to vote, and a timeout. Once created, the `room_controller` agent asks the `personal_assistant` agents to participate in the voting. After all `personal_assistant` agents have voted or the timeout has expired, the `VotingMachine` stops the voting process and determines the winning temperature option.

As can be noted, part of the voting process that was included in the `room_controller` agent code in the previous section is now encapsulated and managed by the `VotingMachine` artifact itself. Another change is that several messages between agents are replaced by interaction of the agents with the `VotingMachine` artifact.

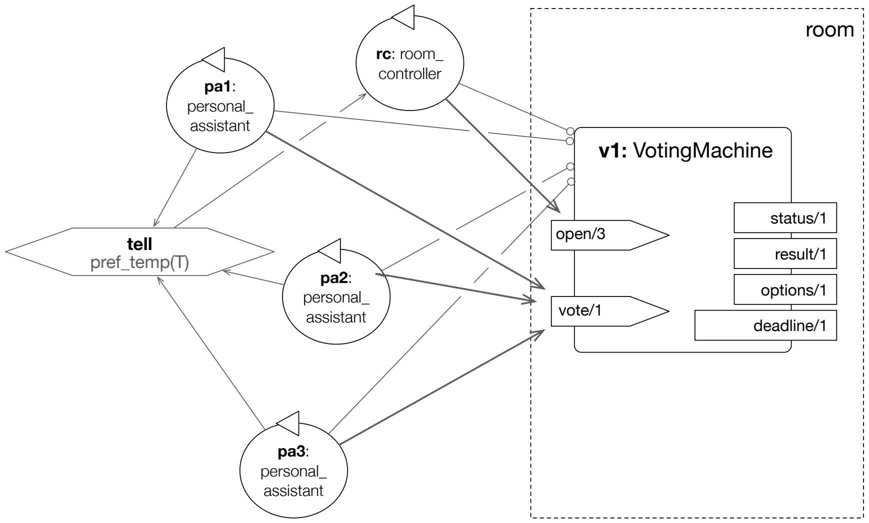


Figure 7.3
The `VotingMachine` artifact to implement the voting protocol. The other artifacts are not represented for the sake of readability.

Programming the `VotingMachine` artifact To program the `VotingMachine` artifact, we have to program the elements of its usage interface by defining its observable properties and its operations. The observable properties are as follows:

- `status` Observable property with value `open` or `closed`, indicating whether votes can still be cast.
- `options` Observable property defined by the `room_controller` agent with the temperature options.
- `deadline` Observable property indicating (in seconds) the remaining time to vote.
- `result` Observable property with the result of the voting process, defined after all votes were cast or after the deadline.

The two following operations are defined:

- `open(Options,Voters,TimeOut)` is used by the `room_controller` agent to open the voting processes to `Voters` (a list of agent names) on a set of options (a list of temperature values); the voting will be open for `TimeOut` milliseconds.
- `vote(OrderedOptions)` enables agents participating in the voting to cast their vote by ordering a set of options according to its own preference; this operation prevents an agent from voting more than once.

When all agents have voted or the deadline is past, the `VotingMachine` artifact determines the winning temperature option according to a particular method and creates the observable property `result` with that value.

We show the following code for the `VotingMachine` artifact. It is a straightforward implementation of the artifact described in figure 7.3 in Java using the features provided by JaCaMo. Note how the `@OPERATION` Java annotation is used to define the `open` and `vote` artifact operations to be made available to agents as actions. Also, `defineObsProperty` is used to create observable properties, and `getObsProperty` is used to retrieve existing observable properties.

```
public class VotingMachine extends GUIArtifact {

    List<String> voters;
    List<Object> votes;
    int timeout;

    public void init() {
        defineObsProperty("status", "closed");
    }

    @OPERATION
    public void open(Object[] options, Object[] voters, int timeout) {
        this.voters  = new ArrayList<>();
        this.votes   = new ArrayList<>();
        ListTerm os = ASSyntax.createList();
        for (Object o: options)
            try {
                os.add(ASSyntax.parseTerm(o.toString()));
            } catch (ParseException e) {
                e.printStackTrace();
            }
        for (Object o: voters)
            this.voters.add(o.toString());
        this.timeout = timeout;

        defineObsProperty("options", os);
        defineObsProperty("deadline", this.timeout);
        getObsProperty("status").updateValue("open");
    }

    @OPERATION
    void vote(Object vote) {
        if (getObsProperty("status").getValue().equals("close")) {
            failed("the voting machine is closed!");
        }
        if (voters.remove(getCurrentOpAgentId().getAgentName())) {
            votes.add(vote);
            if (voters.isEmpty()) {
                close();
            }
        } else {
            failed("you voted already!");
        }
    }

    void close() {
        defineObsProperty("result", computeResult());
        getObsProperty("status").updateValue("closed");
    }
```

Refactoring the agents to act on the `VotingMachine` artifact As shown in the following, the sole modification on the `personal_assistant` agent consists of acting on the `VotingMachine` artifact using the operation `vote` instead of sending its ordered set of options to the `room_controller` agent.

```
+open_voting(ArtName)
   <- lookupArtifact(ArtName, ArtId);
      vm::focus(ArtId).

+vm::options(Options)
   <- ?pref_temp(Pref);
      ?closest(Pref,Options,Vote);
      vm::vote(Vote).
```

namespaces In this code we use the JaCaMo feature of *namespaces*. Namespaces allow the programmer to organize the mind of the agent in several separate spaces or compartments, so that beliefs, events, plans, and actions can be placed together and isolated from others. Each namespace is identified by a name (`vm` in the code above) that is used as a prefix (using `::`) for beliefs, events, and so forth. For instance, the second plan is placed in namespace `vm` and is relevant only for events in this namespace. The focus in line 3 associates observable properties and actions of artifact `ArtId` with the namespace `vm` so that corresponding beliefs and events are placed in that namespace. Operations of that artifact are also placed in the namespace (as used in line 8). Namespaces are features to modularize the agent programming, helping us to avoid name clashes; for instance, different voting machines can be placed in different namespaces and are thus isolated from each other. More details about namespaces can be found in Ortiz-Hernández et al. (2016).

Considering changes in the `room_controller` agent, the plan corresponding to the achievement of the `open_voting` goal is modified to create an artifact instance of the `VotingMachine` artifact, with a unique name corresponding to the conversation ID, with the observable property `options` initialized with the set of options on which to vote, with the list of voters, and with a timeout for agents to vote. After focusing on this artifact, it sends the ID of the artifact to the participating agents (`.broadcast(....)`) so that they can also focus on it. Once the observable property `result` has been created, the `room_controller` agent reacts by generating a new temperature goal as done in the previous step.

```
conv_id(1).

// start a voting protocol whenever a new
// preferred temperature is received
+pref_temp(UT)[source(Ag)]
   <- .println("New preference from ", Ag, " = ",UT);
      // this is only necessary if personal assistant agents do not
      // send untell of previous preferences
      if (pref_temp(Y)[source(Ag)] & UT \== Y) {
         // keeps just the last preference of some agent
         -pref_temp(Y)[source(Ag)];
      }
      !open_voting.

+!open_voting
   <- !get_id(Id);
      .concat(v,Id,ArtNameS);
      .term2string(ArtNameT,ArtNameS)
      .findall(T,pref_temp(T)[source(_)],Options);
      .all_names(AllAgents);
      .my_name(Me);
      .delete(Me,AllAgents,Voters);
      vm::makeArtifact(ArtNameS,"voting.VotingMachine",[],ArtId);
      vm::focus(ArtId);
      vm::open(Options, Voters, 4000);
      .broadcast(tell, open_voting(ArtNameT));
   .

@lId[atomic]
+!get_id(ConvId) : conv_id(ConvId)  <- -+conv_id(ConvId+1).

+vm::result(T)[artifact_name(ArtId,ArtName)]
   <- .println("Creating a new goal to set temperature to ",T);
      .drop_desire(temperature(_));
      !temperature(T);
      .broadcast(untell, open_voting(ArtName));
      //disposeArtifact(ArtId);
```

The plan label (`@lID`) includes the `atomic` annotation. It indicates that no other intention should be executed while this plan is executing. It avoids some race condition while getting a unique identification. The `-+` operator is a short form of a - operation followed by a + operation, as commonly used to update beliefs. In this code, the operation `-+conv_id(Conv_Id+1)` implies the execution of `-conv_id(_)` followed by `+conv_id(Conv_Id+1)`. As previously, the deployment is omitted here because it is similar to the previous cases. Refer to the complete code accompanying the book for details.

Implementing More Complex Coordination Artifacts

Any coordination artifact is a *coordination medium* (Ciancarini 1996), used concurrently by multiple agents, enabling their interaction and managing the dependencies among their actions—resulting in some coordination functionality. In the general case, such management can require the synchronization of the called operations, that is, enforcing some order in their execution of the actions executed by agents. As a simple example, consider a coordination artifact that a team of agents can use to achieve a synchronization point. The idea is that the artifact would provide a `meet` operation, which succeeds only when *all* the agents of the team execute it:

```
public class SyncToolArtifact extends Artifact  {

    private int nTotalFriends;
    private int nFriendsArrived;

    void init(int nFriends) {
        this.nTotalFriends = nFriends;
        nFriendsArrived = 0;
    }

    @OPERATION void meet() {
        nFriendsArrived++;
        await("allFriendsArrived");
    }

    @GUARD boolean allFriendsArrived() {
        return nFriendsArrived == nTotalFriends;
    }

    @OPERATION void reset() {
        nFriendsArrived = 0;
    }
}
```

The artifact is initialized specifying the cardinality of the agent team. A counter `nFriendsArrived` is used to keep track of the number of agents that requested to meet. The counter is incremented each time a `meet` operation is executed. Then, the execution of this operation is suspended until the counter is found to be equal to the cardinality of the agent team. This happens as soon as all agents requested `meet`.

The suspension is enforced by the `await` primitive, which blocks the execution of the operation (releasing the access to the artifact) until the guard (condition) specified as a parameter is evaluated to true. The API allows implementing the guard either as a Boolean method—in that case the name of the method is specified as a parameter of the `await` primitive—or directly as a function/closure, always to be passed as a parameter to `await`. On the agent side, the intention executing the `meet` action is suspended until the operation is completed.

Another primitive that we have already met in a previous chapter, `await_time`, has a similar behavior: it suspends the operation execution until the specified amount of time has elapsed. Overall, the `await` primitive family are expressive enough to realize any kind of synchronization behavior inside artifacts, similar to condition variable mechanisms in monitors used in concurrent programming.

The `VotingMachine` coordination artifact could be extended, for example, with an operation `awaitClosing` that allows synchronizing the agent executing that action with the closing of the voting process. It is worth remarking that synchronization behavior can be often obtained by simply exploiting the observability model of artifacts. In the `VotingMachine` case, for instance, an agent can be aware of the closing by observing the artifact and reacting to changes in the `status` observable property. Nevertheless, there are cases in which it is easier to represent and implement coordinating behavior in terms of actions to be used inside plans, not reactions. In those cases, such synchronization actions can be implemented as operations of properly designed coordination artifacts.

7.4 From Decentralization to Distribution

So far, we consider decentralization of control and responsibilities, using agents (and artifacts) for that purpose, without discussing *distribution*. Distribution accounts for exploiting multiple computational nodes (computers, hosts, and devices)—connected by a network—to execute the system.

In the smart room case study, distribution can be introduced by considering that the building is composed of several rooms. The management of temperature in each room is supported by a `room_controller` with an `hvac`. The owner of the building has defined global preferences for the house. Each `room_controller` takes into account which `personal_assistant` is in the room to manage the negotiated temperature of the room, as presented in the previous versions of our implementation. Coordination among the `room_controller` of the different rooms can be considered so that the global temperature corresponds to the objective of the house. The `personal_assistant` agents are supposed to run on mobile devices, entering and exiting the system dynamically.

When MAOP is used, distribution can be modeled and implemented at two different levels:

- At the agent level, agents are deployed on different nodes using middleware that provides services for direct agent communication among remote agents.
- At the environment level, workspaces are deployed on different nodes, and agents can join and work in remote workspaces.

An important remark here is that in the distributed execution nothing changes, either in the communication model based on speech acts and interaction protocols or in the way agents work with artifacts in artifact-based environments. In the following we discuss two examples about how to program and deploy a distributed MAS using JaCaMo.

Agent Distribution

Although the program of the agents does not change in the distributed execution, that is, agents continue to address other agents by their names, the application file has to be changed to configure how agents are distributed on different nodes. This configuration depends on the particular middleware being used to manage the agent life cycle in the distributed scenario. Here we use JADE as such a middleware, because JaCaMo is already integrated with JADE.

As a concrete example, we consider the extension of the smart room case study to a smart home, in which we have many rooms and thus many `room_controller` agents and participants. To manage all rooms, a new agent, called `majordomo`, is created and communicates with all `room_controller` agents. These agents are distributed as follows: agents of the same room (one `room_controller` and some `personal_assistant` agents) run on a node dedicated to that room; and the `majordomo` runs on its own node. For each node, an application file has to be written and later executed on its host. The `majordomo` application file (`majordomo.jcm`) is as follows (notice the keyword `platform`):

```
mas mj_conf {

    agent majordomo

    platform: jade()  // select JADE as the distribution platform
}
```

When executed, it will launch a JADE platform with the main container and the agent `majordomo` running on it. The application file for a room node looks as follows:

```
// application file for agents running on node of room B210

mas room_b210 {
    agent pa1 : personal_assistant.asl {
        beliefs: preferred("reading", high)
                 preferred("watching", high)
                 preferred("cooking", high)
                 preferred("sport", medium)
        join: room
    }

    // similar for pa2, pa3, ....

    agent rc : room_controller.asl {
        focus: room.hvac
    }

    workspace room {
        artifact hvac: devices.HVAC(20)
    }

    // select and configure JADE as the distribution platform
    platform: jade("-container -host <mdhost>")
}
```

When executed, it will launch JADE with a (not main) container connected to the JADE platform running on the host identified by `mdhost`. Every usual argument used to start JADE can be given as a string parameter of platform `jade(...)`. The concrete internet location for host `mdhost` is provided in the JaCaMo *deployment configuration* file, an application property file informed when running the application file. The content of this file is quite simple, for example:

```
mdhost=host1.my_sweet_home.org
```

The JaCaMo and JADE integration translates Jason KQML messages to JADE FIPA ACL and vice versa as well as using JADE services (as directory facilitator) when necessary. In this case, every Jason agent runs as a JADE agent and can thus send and receive messages from ordinary JADE agents.

Environment Distribution

The environment distribution model provided by JaCaMo basically allows workspaces of an MAS to be executed on different hosts. The mapping between workspaces and hosts can be done either statically or dynamically.

Static mapping In the static case, the mapping is declared as part of the initial configuration of the MAS. The host executing the workspace can be explicitly declared in the application file, in declaring the initial configuration of workspaces, either as a logic name or directly as an IP address (or domain name).

Continuing the smart home scenario, we consider now three workspaces (rooms) – `hallway`, `living_room` and `bath_room` – each one with its own `hvac` and room controller agents. The `hallway` is the parent workspace for both the `living_room` and the `bath_room` (see figure 7.4). The configuration of the workspaces can be specified in the application file as follows:

```
mas smart_home {

  agent majordomo {
    join: hallway
  }

  agent living_room_contr : room_controller.asl {
    goals: temperature(21)
    join:  living_room
    focus: living_room.hvac
  }

  agent bath_room_contr : room_controller.asl {
    goals: temperature(24)
    join:  bath_room
    focus: bath_room.hvac
  }

  workspace hallway {
    host: host1
  }

  workspace living_room {
    artifact hvac: devices.HVAC(15)
    parent: hallway
    host: host1
  }

  workspace bath_room {
    artifact hvac: devices.HVAC(15)
    parent: hallway
    host: host2
  }
}
```

In this case, logic names are used so that `hallway` and `living_room` are declared to be in execution on `host1`, and `bath_room` on `host2`. It is worth noting that if no `host` is specified, workspaces are created by default on the same machine where the MAS is spawned.

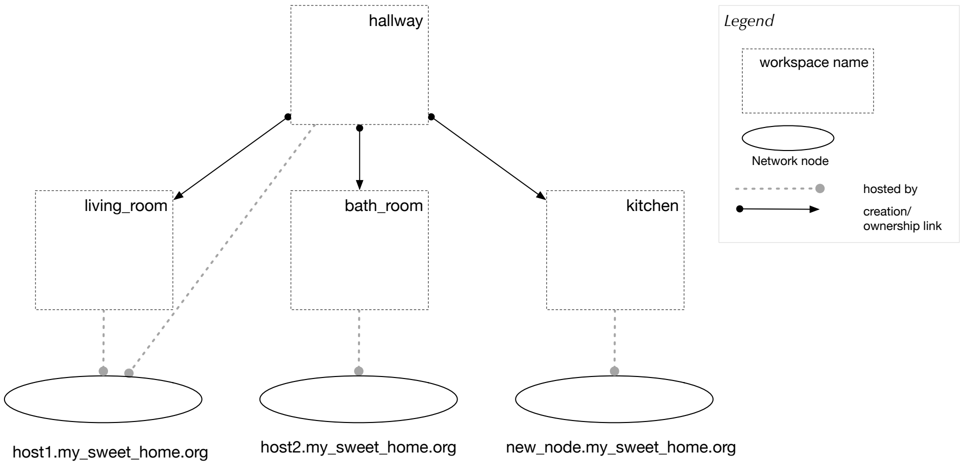


Figure 7.4
Smart house workspaces distributed on different hosts.

The binding between host logic names and some specific IP address or domain name can be specified when the MAS is launched, by means of the separate JaCaMo *deployment configuration* file or directly declaring the properties in the command line. An example of JaCaMo deployment configuration file `smart_home.properties` is

```
host1=host1.my_sweet_home.org
host2=host2.my_sweet_home.org:15000
```

In order for this to work in practice, the JaCaMo infrastructure daemon must be in execution on every host where workspaces are allocated.

As mentioned previously, in order to work with workspaces on remote nodes, an agent doesn't have to specify their physical location, but can refer to their logical name paths. For instance, the `majordomo` agent can join the `bath_room`, which is running on a different host, by executing `joinWorkspace("/hallway/bathroom")`[1] or more simply `joinWorkspace("bathroom")`. The same holds for all workspace operations: for instance the agent can create a new child workspace of `bath_room` by executing `createWorkspace("/hallway/bathroom/tools")`.

Dynamic mapping In some cases, it could be useful to dynamically attach new hosts to an MAS, where to create (and join) workspaces. In those cases, typically the address of the host is known only at runtime, so it is not possible to specify it statically when the MAS is launched. To this purpose, a variant of the `createWorkspace` action is available, which makes it possible to specify as a further parameter the host (address) where to create the workspace. For instance:

```
createWorkspace("/hallway/kitchen","new_host.my_sweet_home.org")
```

As in the static case, the JaCaMo infrastructure daemon must be in execution on that host.

Another main case of dynamic mapping occurs when an agent of an MAS needs to access and work with workspaces of another MAS. For that purpose, the agent needs first of all to *mount* the target workspace (of the other MAS), as in the case of remote file systems. Mounting makes it possible to *link* the remote workspace—possibly running on a different host from the one currently used by the MAS—as a child of some workspace of the MAS. The action available to agents is:

```
mountWorkspace(TargetMAS, TargetWSP, MountPoint)
```

1. The second parameter of `joinWorkspace`, that is, the output parameter representing the workspace identifier, is optional and can be omitted.

where `TargetMAS` is the address of the host serving as entry point of the remote MAS, `TartgetWSP` is the reference (logical path) to the workspace to be mounted, and `MountPoint` is the path to be used locally to access the workspace.

In the smart room (home) case, we can see a clear use for this feature. Each `personal_assistant` agent is running on the mobile device of a human user and so is part of an MAS running on that device. In order to join and work with workspaces of the smart house, a `personal_assistant` agent first needs to mount them on the local MAS. To that purpose, the agent needs to know the address of the host serving as the entry point of the MAS. This could be obtained at runtime, for example, by means of a QR-code or NFC, or a beacon-based system provided by the house. Given the entry point, the agent can execute

```
mountWorkspace("host1.my_sweet_home.org", "/hallway",
                                             "/main/friend\_house")
```

so as to make the `hallway` remote workspace accessible in the local MAS using the path `/main/friend_house` (see figure 7.5).

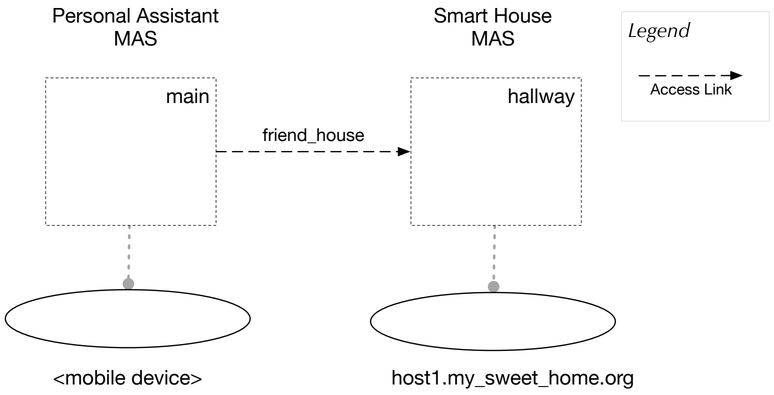


Figure 7.5
Mounting the `hallway` workspace. Making it accessible from the personal assistant MAS by means of an access link (`friend_house`).

The `personal_assistant` agent can then join the remote workspace simply by doing a `joinWorkspace("/main/friend_house").`

Besides the case of mounting, access links can be created between workspaces with the purpose of easing the navigability of the environment. They are similar to soft links used in file systems. By introducing access links, the same workspace can be identified using different paths, besides the ownership/creation links between a parent workspace and its children.

Dealing with Distributed Systems Complexity

The design and development of a distributed system call for dealing with important issues and challenges that are not specifically about multi-agent systems. A main issue is *fault tolerance*, which is the property that enables a system to continue operating properly in the event of the failure of (or one or more faults) some of its components. Another issue is *scalability*, which is the property of a system to handle a growing amount of work by adding resources to the system. We have seen in this chapter that multi-agent systems already provide a level of abstraction that is effective for modeling, designing, and programming either decentralized or distributed systems. Nevertheless, to handle the issues mentioned, MAS technologies and infrastructures need to implement mechanisms and architectural patterns that are typically adopted for engineering robust distributed systems. This aspect is further discussed in chapter 11.

7.5 What We Have Learned

In this chapter we dived into agent interaction, exploring approaches for programming and coordinating interaction among agents. These approaches span from direct communication based purely on speech acts and interaction protocols to environment-mediated coordination, based on coordination artifacts:

- Direct agent-to-agent communication used, for example, to share beliefs and to delegate goals;
- Interaction protocols used to coordinate communicative actions among agents in structured patterns of interactions, implemented in the agents themselves or implemented in an artifact leading to the notion of coordination artifacts; and
- Coordination artifacts, that is, artifacts situated in the environment not only for encapsulating physical resources but also for structuring and managing interaction patterns among agents.

We emphasize that the two alternative solutions shown here for the agreement on a target temperature differ significantly. As discussed previously, the message-based approach requires a lot more communication among the agents but is completely decentralized, while the artifact-based approach requires less message exchange but introduces a centralized mechanism via an artifact that is shared by all agents to manage the interaction protocol, leaving the decision and strategies decentralized in the agents. In the next chapters we consider a further approach based on *organization* abstractions, which allows for a more abstract and high-level approach to specify coordination in multi-agent programs.

Besides management and coordination of interactions among agents, we also have extended our knowledge on programming:

- Agents
  - execution of concurrent plans corresponding to different intentions, in this example the computation of target average temperature and management of the HVAC; and
  - how to use test goals and how to handle them.
- Environment
  - the use of multiple workspaces.
- Execution
  - agent types that can be instantiated in several agents tuned with specific beliefs and/or goals specified in the application file; and
  - simple distributed JaCaMo programs.

Exercises

Exercise 7.1 Implement a ping-pong MAS with indirect interactions: Agent A executes *play* on artifact Left, creating signal *A played* perceived by agent B focusing on Left; B then executes *play* on artifact Right, creating signal *B played* perceived by agent A, and so forth. Agent C (the controller) focuses on Left and Right and counts the number of times it perceives the signals (A played, B played).

Exercise 7.2 Implement a ping-pong MAS with indirect and direct interactions. In particular, extend the code of the preceding exercise so that when agent C has counted 10 (A played, B played) sequences, it sends a message *stop* to agents A and B, who then have to stop playing.

Exercise 7.3 Develop new strategies for handling incoming requests for temperature change as managed in this chapter (e.g., waiting some amount of time for new requests before actually proceeding to change the target temperature, or waiting until there are at least a few new requests).

Exercise 7.4 Currently the `personal_assistant` agent does not observe the `hvac` artifact, meaning that it does not know the current temperature. Change the code of this agent to get this information. Then change the code so that the agent either sends again its preferred temperature if the temperature fails to change within a reasonable time interval or propose its user to quit the room (in this case, adapt the `UserGUI` artifact so that the agent can act and change a textfield).

Exercise 7.5 Reimplement the `room_controller` plan that computes a unique conversation ID using an artifact that produces unique IDs, so that the `atomic` execution of the intention is not required anymore.

Exercise 7.6 Instead of the voting mechanism, implement an alternative consensus technique in all versions of the room scenario developed in this chapter (based on direct and indirect communication). Evaluate the effort required for creating each of those versions.

Exercise 7.7 Distribute the execution of the agents and workspaces of previous exercises and evaluate the required programming effort.

Exercise 7.8 Create a `personal_assistant` agent without plans for voting. When this agent is asked to vote, it asks a roommate for a voting plan and then votes using that same strategy.

8 The Organization Dimension

We saw in previous chapters how the agent and environment dimensions provide the basic elements for building multi-agent systems as a set of individual autonomous agents communicating and working in shared environments. In this chapter, we complete the MAOP picture by considering the *organization dimension*, which provides the concepts and first-class programming abstractions to specify and govern *complex* MAS from a *macro* perspective, compared with the micro (individual-based) one offered by the two other dimensions. We start with a global overview, addressing the overall map of abstractions offered to those who want to program an organization in a multi-agent system. The importance of this dimension is stressed with respect to the *multi-agent oriented programming* point of view developed in this book. Once all concepts and abstractions have been explained and positioned relative to each other, we discuss the organization execution model, in which further organization-related abstractions are introduced. We finish the chapter with notes and historical elements so that the interested reader can get deeper knowledge on approaches directly or indirectly related to organizations and their use in the area of multi-agent system.

8.1 Overview

Organization is a broad concept that has different meanings in several domains ranging from social and management sciences to computer science. In the multi-agent research domain alone, several proposals of organizational models exist (see Aldewereld et al. [2016]; Coutinho et al. [2009] for surveys on some of them). Each of them highlights specific properties, emphasizing either descriptive or prescriptive models related to the structuring, the coordination, or the regulation of the agents working together in a shared environment.

To better understand the viewpoint taken in talking about organization, we go back to the human work environment shown in figure 5.1. In this figure, we can describe the activity taking place among the agents by abstracting from them and identifying a main group `bakery_staff` in which the global activity of the agents takes place. In the context of a group, agents can be considered to play some roles. For instance, the agent `helen` is playing the `pastry_chef` role, and `john` is playing the `assistant` role. Being in tight interaction together, they build a `cake_staff` subgroup of the main group. In another part of the workspace, another set of agents form the codescheduling_staff subgroup and are in charge of realizing the schedule of the day: the agent `mary` plays the `manager` role; `bob`, `henry`, and `anna` help her to build the schedule: they play the `planner` role. The agent `paul` is playing two roles: the `planner` role in the `scheduling_staff` and the `archivist` in the main group because it interacts with the `cake_staff` subgroup to retrieve cake recipes and the `scheduling_staff` for storing the produced schedule. In each of these groups and according to their roles, the agents are expected to coordinate with each other while executing part of the predefined plans. For example, the plan `wedding_cake_recipe` assigned to group `cake_staff` is to be executed following the recipe for making a wedding cake, and `week_schedule` in `scheduling_staff` is to be executed in order to plan the work to be done over the week).

As illustrated in this example, organizations deal with the modeling of *supra-individual phenomena* (Gasser 2001). The corresponding models consist of structured cooperation patterns going beyond the activity of a single agent operating in the multi-agent system. The concepts (e.g., roles and groups) involved in the definition of the cooperation patterns are thus distinct from those belonging to the agent or environment dimensions discussed in the previous chapters. Similar to what exists in management science (Malone 1999), organizations structure and aid in the *decision* and *interaction* of the agents to fulfill tasks and achieve goals in the environment while guaranteeing a global coherent state of the system. With regard to sociology (Bernoux 1985), organizations may express division of tasks into subtasks, distribution of roles, and assignment of authority among the agents participating in the organization. More generally, they may concern the structuring of knowledge, culture, history, and capabilities to be shared and used by agents.

Organization specification and organization entity Organizing a multi-agent system is a process that starts with a *definition* phase undertaken by the stakeholders involved in the development of the MAS, followed by an *execution* phase in which the agents behave under the constraints imposed by the specified organization. This process may also consist of an iterative interleaving of these two phases, undertaken by the agents themselves through reasoning on their collective behavior. The agents define and adapt their organization while acting, perceiving, and cooperating in it. This organizing process produces two descriptions: an *organization specification* and an *organization entity*.

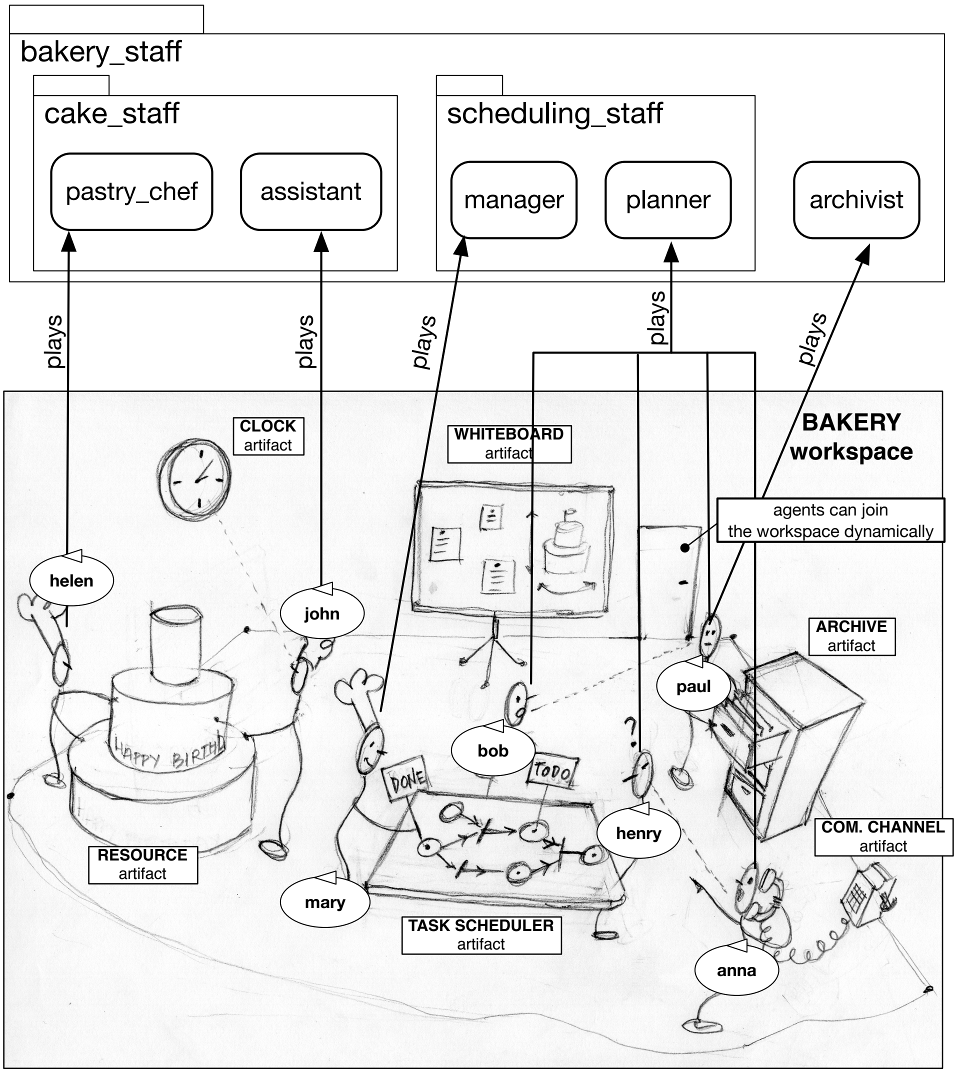


Figure 8.1
Organization in the bakery workshop scenario.

organization specification

An *organization specification* is a declarative description (Van-Roy and Haridi 2004) answering a *what* question, that is to say, the expected behavior to be produced by the agents, without explaining *how*, that is to say, the actions needed to achieve the results. Those choices on courses of action pertain to the agent level. For instance, in the use case shown in figure 8.1, the organization specification states that the `bakery_staff` group is composed of two subgroups, `cake_staff` and `scheduling_staff`, in which the `pastry_chef`, `assistant`, `manager`, `planner`, and `archivist` roles can be played by the agents. Although not shown in the figure, the `cake_staff` group can be responsible for the `wedding_cake_recipe` and `pudding_cake_recipe` collective plans to be undertaken by the agents according to the roles played in that group. As can be noted, whereas the organization specification defines one role `planner` in the `scheduling_staff`, several agents can play this role in the group entity created from the `scheduling_staff` definition. In the same way, depending on the situation (e.g., baking several cakes), there could exist in the organization entity several group entities created from the `cake_staff` group definition to coordinate the agents according to the recipes under their responsibility.

organization entity

An *organization entity* corresponds to the enactment of the organization specification by the agents. Whereas the organization specification defines the expected behavior of the agents, the organization entity describes the evolving state of their coordinated and regulated behavior in relation to the expected one. These include, for example, the various created groups and the roles that agents chose to play in each group. Considering again the case shown in figure 8.1, the organization entity enacting the organization specification described above states that agent `john` is playing role `pastry_chef` in group `wedding_cake`, agent `mary` is playing role `manager` in group `week`, and so forth.

While observing the agents executing in a multi-agent system, one can notice relations and cooperation patterns that are not represented in the organization entity; that is to say that agents may coordinate with each other without referring to the organization. One can also observe cooperation patterns in the organization entity that do not have any counterpart in the organization specification. Different from the previous observation, it often means that agents are misbehaving with respect to the organization to which they belong. That is to say that they may be *violating* some expected behavior of the organization. For instance, an agent playing `assistant` role and expected to execute tasks related to the `wedding_cake` recipe might be violating the specification by preparing marzipan when the recipe forbids such a task because it calls for whipped cream on the cake. Because an organization regulates the behavior of the agents operating in it, depending on the gravity of the violations, agents may be *sanctioned*.

The organization entity changes all along the execution of the system, updated each time the agents create or delete groups, adopt or leave roles, and so forth. The changes to the organization specification may be less frequent because they are costly and with deeper consequences. The approach described in the book focuses on the programming, in particular the definition of both organization specification and organization entity based on the concepts available in the organization dimension.

Multi-faceted dimension Descriptions of organization may cover several facets of the collective activity of the multi-agent system (Coutinho et al. 2009). In this book, the concepts in the organization dimension allow the modeling and programming of

- the structure of the organization in terms of *roles*, *links*, and *groups* (called hereafter *structural abstractions*);
- the coordination in terms of *missions*, *plans*, and *goals* (called hereafter *functional abstractions*); and
- the regulation in terms of *norms* to constrain the autonomy of the agents with respect to their structuring and coordinated activities (called hereafter *normative abstractions*).

As we discuss subsequently in this chapter, whereas structural and functional abstractions are independent sets of concepts, the normative abstractions bind them together.

Programming organizations As introduced previously and stressed in the chapters about the other two dimensions, concepts belonging to the organization dimension are first-class entities that are clearly distinct from those of the agent and environment dimensions. They form the basis for a customized tag-based language based on extensible markup language (XML) to express the elements of the *organization specification*. For example, in figure 8.2, the `bakery_staff` definition contains the definition of roles as well as the definition of the subgroups `cake_staff` and `scheduling_staff`, and the definition of the collective plans that we call schemes, such as `wedding_cake_recipe`. They are also the basis for a language based on predicates and functions to represent the organization entity within the agents as beliefs (Hübner et al. 2007), as well as organizational facts to regiment or enforce the behavior of the agents in the system against the constraints stated by the organization specification (Hübner et al. 2011). These representations deal with the created groups, the roles played by agents in these groups, the parts of plans undertaken by the agents in the context of their missions, the active norms that regulate the agents' behavior, and so forth.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="http://moise.sourceforge.net/xml/os.xsl" type="text/xsl" ?>

<organisational-specification
    id="bakery"
    os-version="0.8"

    xmlns='http://moise.sourceforge.net/os'
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xsi:schemaLocation='http://moise.sourceforge.net/os
                        http://moise.sourceforge.net/xml/os.xsd' >

  <structural-specification>
    <role-definitions> <role id="pastry_chef"/> ... </role-definitions>
    <group-specification id="bakery_staff"> ...
            <group-specification id="cake_staff"> ... </group-specification>
            <group-specification id="scheduling_staff"> ... </group-specification>
            ...
    </group-specification>
  </structural-specification>
  <functional-specification>
    <scheme id="wedding_cake_recipe"> ... </scheme>
    <scheme id="pudding_cake_recipe"> ... </scheme>
    <scheme id="week_schedule"> ... </scheme>
  </functional-specification>
  <normative-specification>
    ...
  </normative-specification>
</organisational-specification>
```

Figure 8.2
General view of the XML organization specification for the bakery use case.

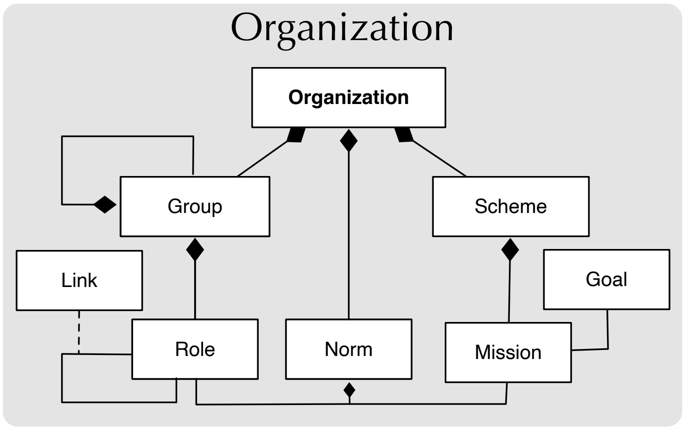


Figure 8.3
Main concepts in the organization dimension.

As can be imagined from this discussion, *reorganization*, that is, changing the organization, may be carried out at the organization entity or organization specification levels. Interestingly, such changes may be made by the agents themselves. Deciding on their own or collectively, they can change, for instance, the assignment of roles to agents keeping the same organization specification. They can also change the organization specification by defining new roles in a group, for example, changing the mission definitions or the norms themselves. Changing the organization specification implies changing the organization entity and thus the MAS expected overall behavior.

8.2 Organization Abstractions

This section details the abstractions available in the organization dimension to define the structural, functional, and normative facets of an organization, used to define the corresponding specifications and entities.

Structural abstractions The structural abstractions address individual, social, and collective concerns defining the structure of an organization (see figure 8.3):

- the *role* concept is used as a programming abstraction of the position that individual agents can occupy in the structure of an organization;
- the *link* concept is used as a programming abstraction of the type of interaction that can take place among agents in a group when playing the corresponding roles; and

- the *group* concept is used as a programming abstraction of a possible community of agents that can exist in the structure of the organization.

Groups are the entry points of the agents into an organization: by adopting a role, an agent enters a group to which the role belongs and becomes connected to other agents playing roles linked to its role. An agent can participate in several groups under various roles.

```
<structural-specification>

<role-definitions>
    <role id="role0"/>
    <role id="role1"> <extends role="role0"/> </role>
    <role id="role2"> <extends role="role0"/> </role>
</role-definitions>

<group-specification id="group1">
    <roles>
        <role id="role1" min="1" max="2"/>
        <role id="role2" min="1" max="1"/>
    </roles>

    <links>
       <link from="role1" to="role2" type="authority"
             scope="intra-group" bi-dir="false" />
       <link from="role0" to="role0" type="communication"
             scope="intra-group" bi-dir="true" />
    </links>
    <formation-constraints>
       <compatibility from="role1" to="role2" bi-dir="true"/>
    </formation-constraints>
</group-specification>
</structural-specification>
```

Figure 8.4
Example of a simple structural specification.

structural specification

The *structural specification* (see figure 8.4) defines templates of organizational structures that are enacted by the agents in organization entities. A structural specification is composed of

role

- a list of role definitions in which each *role* is identified by a label that is unique in the specification (e.g., `role0`, `role1`). An *inheritance relation* among roles may complement this definition (e.g., `role1` and `role2` inherit from `role0`). It enables the reuse of properties attached to the inherited role. As discussed subsequently, properties of a role are the links and constraints in which it is involved as well as the normative constructs to which it belongs. By default, all roles inherit from the predefined `soc` root role.

group

- a hierarchy of groups in which each *group* is identified with a unique label in the organization (e.g., `group1`). It is composed of roles picked from the list of defined roles (`role1` and `role2`), links between these roles (e.g., the authority link between `role1` and `role2`), possibly other subgroups, and a set of group-formation constraints.

link

A *link* is a labeled relation connecting two roles together within the group. The current set of labels comprises *communication*, *authority*, and *acquaintance*. They allow the definition of three interaction networks among roles: *communication* networks stating who can interact by communication with whom while playing the corresponding roles, *control* networks stating who has authority over whom, and *acquaintance* networks expressing who can represent and access information from whom.

group-formation constraint

The *group-formation constraints* define expected properties on the structure of the organization entity built from the organization specification. They concern the roles, links, and subgroups participating in the definition of the group:

– A *role-compatibility constraint* is a directed relation between two roles of a group. It enables an agent to adopt the target role while already playing the source role (by default roles are incompatible, that is, they cannot be played by the same agent in the organization). For instance, in figure 8.4, `role1` and `role2` are set as compatible.

– A *role-cardinality constraint* defines upper and lower bounds on the number of agents that can play the role in the corresponding group entity (e.g., at least one and at most two agents should play `role1`).

– A *group-cardinality constraint* defines upper and lower bounds on the number of subgroup entities that can be created from the subgroup defined within the group.

The structural specification defines thus the template structure of an organization entity as a nonoverlapping hierarchy of groups and subgroups, in which each group is composed of roles, links, possibly subgroups, and group-formation constraints. All elements of the structural specification are thus part of a group definition. For instance, a link can connect two roles appearing in two different group definitions if and only if there exists a supergroup of the two groups including the definition of such a link.

Agents participate in an organization entity by creating *group entities* following the template structure and group-formation constraints of the structural specification. According to the group definition, agents may adopt one or several roles, interact with each other according to the corresponding links between roles, and create subgroups. Depending on the group cardinality, one or several group entities referring to the same group definition may exist in the organization. As shown in figure 8.5, two group entities are enacted from the same structural specification. As we can see, whereas group entity `g1` (following specification `group1`) contains

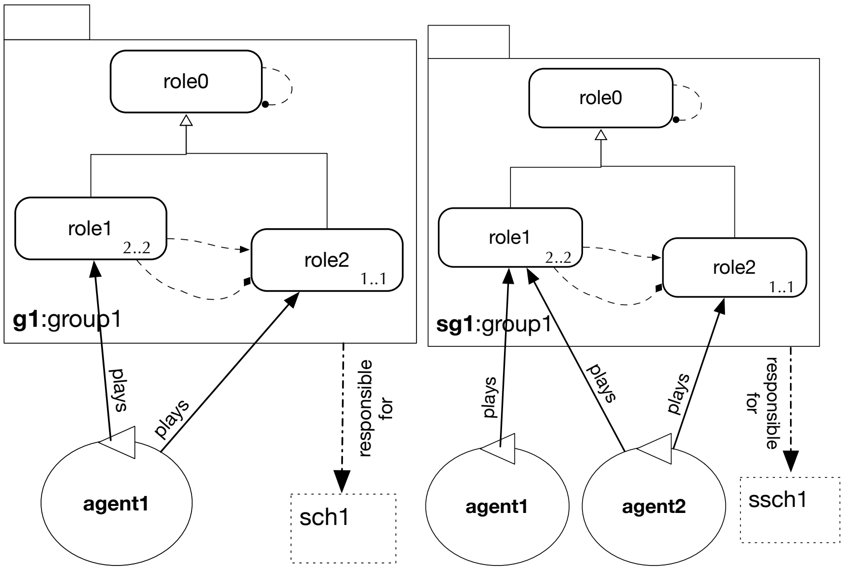


Figure 8.5
Group entities enacted from the structural specification defined in figure 8.4.

only one agent playing both roles, group entity `sg1` (following the same specification) contains two different agents playing roles `role1` and `role2`. If the compatibility link did not relate these two roles, both group entities would not be well formed, violating the default incompatibility constraint between roles in a group. In the same way, either `agent1` or `agent2` would not be able to play role `role1` if the maximum cardinality of this role in the definition of group `group1` was equal to 1. We see subsequently in this chapter how agents can enact such group entities using organizational actions. As shown in figure 8.5, the group entities `sg1` and `g1` are both connected by a *responsible for* link to `ssch1` or `sch1` social scheme entities. These two entities are created from the functional abstractions described next.

Functional abstractions As for the structural abstractions, the concepts related to this facet (see figure 8.3) address individual, social, and collective concerns of the behavior of the agents within an organization. They are

- the *organizational goal* concept abstracts a state of affair that has to be satisfied by one or several agents;
- complementing goals, a *mission* gathers goals that have to be achieved under the responsibility of an individual agent;

- a *social plan* at the organizational level denotes a structure of interrelated organizational goals to be satisfied by multiple agents that have to coordinate with each other to handle the interdependencies between goals. Such plans may be set either by the MAS designers who use their expertise to define the plan or by the agents themselves, for instance, by keeping track of their (best) past solutions or by planning; and
- a *social scheme* gathers a social plan with its organizational goals and corresponding missions. It denotes the collective and coordinated behavior that is expected to be produced by a group of agents in the organization.

The organizational goal and social plan concepts are central to defining the expected behavior within an organization. Using goal as a primitive construct instead of action allows fewer constraints on the agents. The organization is more interested in the satisfaction of the state of affairs denoted by the goal than in the means (i.e., the particular actions) used to reach this state of affairs.

```
<functional-specification>
    <scheme id="scheme1">
        <goal id="goal1" ds="description of goal1">
            <plan operator="sequence">
                <goal id="goal2" ttf="20 minutes" ds="description of goal2"/>
                <goal id="goal3" ds="description of goal3"/>
                <goal id="goal4" ds="description of goal4"/>
            </plan>
        </goal>

        <mission id="mission1" min="1" max="2">
            <goal id="goal2" />
            <goal id="goal4" />
        </mission>
        <mission id="mission2" min="1" max="1">
            <goal id="goal3"/>
        </mission>
    </scheme>
</functional-specification>
```

Figure 8.6
Example of a simple functional specification.

The *functional specification* (see figure 8.6) gathers templates of coordinated behavior defined as a list of social schemes in which each *social scheme* is identified by a unique label in the functional specification (e.g., `scheme1`) grouping together

functional specification

social scheme

- a goal decomposition tree where the root is a global goal and the leaves are goals that can be satisfied by the agents. A *goal* is denoted by an identifier (e.g., `goal1`, `goal2`). A *goal cardinality* constrains the number of agents that are in charge of achieving or maintaining the goal. Each non-leaf goal is decomposed into subgoals by *plans* using three operators:

goal

plans

- sequence ",": the plan "$g_1 = g_2, g_3$" means that goal g_1 will be satisfied if and only if goal g_2 and subsequently goal g_3 are satisfied;
- choice "|": the plan "$g_1 = g_2 | g_3$" means that goal g_1 will be satisfied if one and only one of the goals g_2 or g_3 is satisfied; and
- parallel "||": the plan "$g_1 = g_2 || g_3$" means that goal g_1 will be satisfied if both g_2 and g_3 are satisfied, and they can be pursued in parallel.

In our example, goal `goal1` is decomposed as a sequence of the three goals `goal2`, `goal3`, and `goal4`. They can be complemented with *dependence relations* among goals expressing that the achievement of a goal depends on the achievement of another goal.

mission
- a list of missions in which each *mission* gathers goals that may be assigned to an agent participating in the organization. A label identifies it uniquely in the scheme (e.g., `mission1`, `mission2`). When an agent participates in an organization entity, it *commits* to missions, meaning that it will try to achieve all the goals contained in the mission. For instance, in the case of `mission1`, `goal2` and `goal3` have to be satisfied. A *mission cardinality* constrains the minimum and maximum number of agents that can commit to a mission (e.g., at least one and at most two agents for mission `mission1`).

The functional specification of an organization is defined as a set of social schemes. No missions or goals can be defined outside a social scheme.

As shown in figure 8.7, two different *scheme entities* are created and set under the responsibility of the group entities presented in figure 8.5. Whereas `agent1` is committed to `mission1` and `mission2` in the first scheme entity, both agents `agent1` and `agent2` are committed to `mission1` in the second scheme entity. From these commitments, it is expected that agents will achieve the goals involved in the definition of the missions. The commitment of agents to missions is not done at random or at the agent's will. It follows the norms defined in the normative specification, the last facet of the definition of the organization specification, discussed next.

Normative abstractions Whereas the structural abstractions address the structuring of the agents in the system and the functional abstractions target the coordination of their behavior, the normative abstractions (see figure 8.3) are concerned with the *regulation* of the agents' behavior in the organization. The main concept is *norm*: A norm defines the rights and duties of agents, taking the viewpoint of its autonomy in the organization. To that aim, it connects together structural and functional abstractions with deontic modalities, to express what is obliged, permitted, or prohibited and for which agents (while they play the roles in that organization). As we can see, contrary to other normative approaches in the literature,

norm

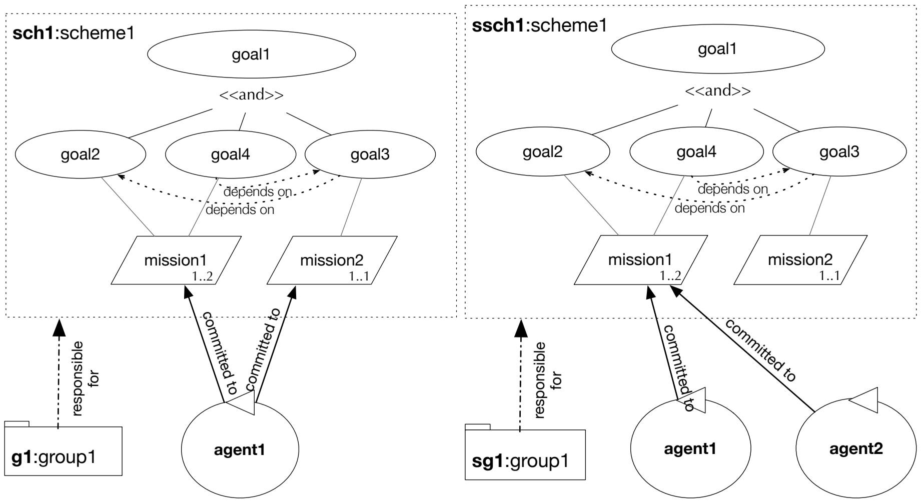


Figure 8.7
Social scheme entities enacted from the functional specification defined in figure 8.6 in accordance with the group entities represented in figure 8.5.

norms are defined in the context of an organization. This is an important difference with regard to other normative approaches that define norms independently of any structure or coordinated functioning.

```
<normative-specification>
    <!-- the norms of the application -->
    <norm id="norm1" type="obligation"
          role="role1" mission="mission1"/>
    <norm id="norm2" type="permission"
          role="role2" mission="mission2"/>
</normative-specification>
```

Figure 8.8
Example of a simple normative specification.

The *normative specification* (see figure 8.8) is composed of a list of *abstract norms* that are expressed as follows: *normative specification*

- a *deontic modality*, which can be an *obligation* or a *permission* *deontic modality*
- the role that the deontic modality refers to; the given role identifier, defined in the structural specification (e.g., `role1`, `role2`), is the *bearer* of the deontic modality; *norm bearer*
- the *mission* that the deontic modality refers to (e.g., `mission1`, `mission2`). *norm mission*

norm activation condition

- an optional *activation condition* that expresses the condition under which the norm is considered to be active. It is used to check some properties of the organization entity or the status of active norms in the organization (e.g., fulfilled, unfulfilled). When not defined, it is considered as being true by default, meaning that the corresponding norm is activated as soon as the organization entity is created; and

time constraint

- an optional *time constraint* that defines a temporal expression stating a time constraint on the fulfillment of the norm. After this deadline, the norm is considered unfulfilled.

These norms are abstract because they express expected behavior in the abstract terms of role in a group for a mission in a social scheme under the responsibility of the group. After enactment of the organization specification, they are interpreted and considered in the context of the organization entity; particular agents playing that role and particular missions are interpreted in the context of the actual group entity. The norm is thus assigned to any agent playing that role.

As shown in figure 8.7, the agents are committed to missions `mission1` and `mission2` according to what the norms state and to the definition of the social scheme and group entities: `agent1` and `agent2` are committed to `mission1` because both play role `role1`, which is the bearer of an obligation to commit to mission `mission1` according to norm `norm1`, and `agent1` is committed to `mission1` and `mission2` in the other organization entity because it is playing `role1` and `role2`.

Partial Specifications of an Organization

Even if the definition of structural and functional specifications is not mandatory, missing one or both introduces a burden and extra computation in the agents.

If the structural specification is missing, there is no structure to help the agents to cooperate. In the case shown in figure 8.1, they should negotiate among themselves to define who has the lead on the schedule definition and with whom to cooperate since they do not know who is involved in the collective task of schedule definition.

If the functional specification is missing, the agents have to reason about a collective plan every time they want to act together. For instance, in the case shown in figure 8.1, they have to (re)define a recipe for a cake and coordinate among themselves (e.g., through the exchange of communication messages) each time they have to prepare a cake. Even with a small search space of possible plans (because the structure constrains the agents' options) this may be a hard problem. Furthermore, the plans developed for a particular problem may be lost (e.g., if the agents leave the organization), because there is no organizational memory in which to keep these plans.

In the case that both specifications are missing, the problems are the union of the above drawbacks.

Besides avoiding such problems, the advantage of having the three specifications defined is that the agents have more information to reason about others' positions in the organization and thus better interact with them. As shown in (Hübner et al. 2004), having the structural and functional specifications independent helps in the definition of a flexible reorganization process: the MAS can change its own structure without changing its coordination, and vice versa. For instance, in the case shown in figure 8.1, the same structure of group `cake_staff` can be kept while the MAS executes in a coordinated manner various cake recipes.

8.3 Organization Execution

organization specification life cycle

We now describe the life cycle of an organization. The *life cycle of the organization specification* depends either on actions of the stakeholders who develop the organization of the multi-agent system (e.g., changes of functional or nonfunctional requirements) or on actions of the agents themselves engaging in a reorganization process (Hübner et al. 2004). Below, we put aside this latter life cycle and focus on the *life cycle of the organization entity*, which is encountered in any execution of a MAS.

organization entity life cycle

As introduced in the previous section, the *organization entity* represents the state of the organization in terms of organizational statements defined from the structural, functional, and normative concepts as well as data structures appearing in the corresponding specifications of the organization specification. This representation is distributed into three types of entities: group, social scheme, and normative entities.

group entity

A *group entity* is related to a group defined in the organization specification. The state that it represents contains the owner of the group entity, the links to children and parent group entities, and the set of role-player agents (i.e., a set of ⟨agent, role⟩ tuples) with their links to other role-player agents. A group entity manages this representation in coordination with the parent or child group entities, taking into account the group-formation constraints defined in the structural specification. Several group entities referring to the same group specification may exist in an organization entity. The structural state of an organization entity is built from the distributed representations managed by each group entity. It represents how agents populating the organization are structured into groups, which roles they play, and links along which they interact with each other.

social scheme entity

A *social scheme entity* is related to a social scheme defined in the functional specification. The state that it represents contains the owner of the social scheme entity[1], the group entities responsible to provide agents for the scheme execution, the commitments to missions, and the state of each goal appearing in the plan. The social scheme entity manages this state, taking into account the constraints defined in its specification. With regard to group entities, several social scheme entities referring to the same social scheme definition may exist in the organization entity. The functional state of an organization entity is built from the distributed representations, managed by each social scheme entity, of the status of the missions and goals under consideration by the agents according to the functional specification.

normative entity

A *normative entity* is related to group and social scheme entities. Created every time a group entity becomes responsible for a scheme entity, it can also be created by the agents to manage a particular set of norms. The normative entity contains the status of a set of norms built from the abstract norms of the normative specification, in particular those in which the role and mission are related to some corresponding group and social scheme entities. The management of the normative entity takes into account the evolution of the group and social scheme entities. Like other entities, several normative entities may refer to the same set of abstract norms of the normative specification. The normative state of an organization entity is built from the distributed representations, managed by each normative entity, of the status of the norms.

From this discussion, we see that the management of an organization entity is distributed in several entities: group, social scheme, and normative entities. The consistency of these distributed representations is ensured by coordinated management of the group/subgroup entities, group and social scheme entities, and normative entities with group and social scheme entities. As we subsequently see, each of these entities has its own life cycle. The life cycle of an organization entity results thus from the tight interaction among the life cycles of each of these entities.

organization entity life cycle

Organization entity life cycle The *life cycle of the organization entity* is as follows:

1. Creation of the organization entity on the basis of a chosen organization specification (see lines 9–12 in the code that follows, which corresponds to an agent

1. Note that the owner of a group (resp. social scheme) entity is not necessarily the creator of the group (resp. social scheme) entity. Immediately after the creation, there is no owner; once created, any agent can become the owner. When it has an owner, only the owner can change the ownership.

acting to create an organization entity). It consists then of sequences of creation of group, scheme, or normative entities participating in the organization entity.

2. Execution of the life cycle of the group, social scheme, and normative entities in relation to the life cycle of the agents and environment. It consists of continuous updates and changes produced by the adoption of roles, creation of groups, commitment to missions, fulfillment or not of norms, and so forth, according to the organization specification that rules agent behavior. More details about each entity life cycle are described in the sequel.
3. Destruction of the organization entity in the MAS as soon as all the other entities belonging to it have been deleted.

Group entity life cycle The *life cycle of a group entity* has the following steps:

group entity life cycle

1. Creation of a group entity based on a group definition (lines 15–16 use the `createGroup` operation to add a group entity `g1` to the `smorg` organization entity). The group entity is connected to its parent according to the group/subgroup hierarchy defined in the structural specification. The root group entity of the hierarchy should be the first one created.
2. Role adoption by agents who become role players in the group entity (lines 18–20), with the use of the `adoptRole` operation stating which role is adopted in the group entity `GrArtId`.
3. Ensuring that the group entity is well formed, that is, that the group-formation constraints are satisfied (`.wait` in line 22 stops the execution of the intention and waits for the signal `formationStatus(ok)` generated `GrArtId` when it is well formed). It means that the group/subgroup entities hierarchy is well formed, and that the subgroup entities are well formed, and that the group entity has a valid structure of interconnected role players with a valid number of agents playing them (there is no role-compatibility problem, and the number of agents is greater than or equal to the minimum and less than or equal to the maximum stated by the role cardinality).

well-formed group entity

4. Assignment of the group entity to be responsible for the execution of one or several social scheme entities (line 26 uses `addScheme` operation for that). Being responsible for a social scheme means that the agents in the group entity are the agents that have to commit to the missions of the social scheme. For instance, if we have two social scheme entities (s_1 and s_2) of the same scheme specification (sch_1) and two group entities (g_1 and g_2), such that s_1 (resp. s_2) is under the responsibility of g_1 (resp. g_2), only agents playing roles in g_1 can (or have to) commit to missions in s_1 and only agents from g_2 can (or have to) commit to missions in s_2. For each social scheme that a group entity becomes

responsible for, a normative entity is created and connected to the corresponding social scheme entity and group entity.

5. Disconnection of the social scheme entities (and corresponding normative entities) that no longer have agents committed to one of their missions.
6. Deletion of the group entity when it has no subgroups connected to it and no social scheme responsibility, and all the agents participating in it have left their roles.

```
7   +!start : true
8      <- // Creation of organization entity "smorg"
9         createWorkspace(smorg);
10        joinWorkspace(smorg,WspId);
11        makeArtifact(smorg, "ora4mas.nopl.OrgBoard",
12                     ["src/org/org.xml"], OrgArtId)[wid(WspId)];
13        focus(OrgArtId)[wid(WspId)];
14        // Group-entity lifecycle: Creation of group entity "g1" in "smorg"
15        createGroup(g1, group1, GrArtId)[artifact_id(OrgArtId)];
16        focus(GrArtId)[wid(WspId)];
17        // Group-entity lifecycle: Adoption of role "role1" in "g1"
18        adoptRole(role1)[artifact_id(GrArtId)];
19        // Group-entity lifecycle: Adoption of role "role2" in "g2"
20        adoptRole(role2)[artifact_id(GrArtId)];
21        // Group-entity lifecycle: Waiting that "g1" is well-formed
22        .wait(formationStatus(ok)[artifact_id(GrArtId)]);
23        // Scheme-entity lifecycle: Creation of social scheme entity "sch1"
24        createScheme(sch1, scheme1, SchArtId)[artifact_id(OrgArtId)];
25        // Group-entity lifecycle: Adding of "sch1" under the responsibility of "g1"
26        addScheme(sch1)[artifact_id(GrArtId)];
27        focus(SchArtId)[wid(WspId)];
28        // organization entity "smorg" created
29        .
30
31  // Scheme-entity lifecycle: commitment to missions
32  +permission(Ag, MCond, committed(Ag, Mission, Scheme), Deadline) : .my_name(Ag)
33     <- commitMission(Mission)[artifact_name(Scheme)].

46  // Common definitions pertaining to Scheme-entity and Normative-entity lifecycle
47  // Obedience to norm prescriptions
48  { include("$moiseJar/asl/org-obedient.asl") }
```

social scheme entity

Social scheme entity life cycle The *life cycle of a social scheme entity* has the following steps:

1. Creation of the social scheme entity on the basis of a social scheme defined in the functional specification (line 23) with the use of the `createScheme` operation on the `smorg` organization entity.
2. Once the social scheme entity is under the responsibility of a group entity, agents playing a role in the group entity responsible for the scheme entity are permitted or obliged to commit to the missions defined in the social scheme according to the norms defined in the normative specification (see line 48 where common plans to commit to missions are included).

well-formed social scheme entity

3. Once the well-formedness of the social scheme entity is determined (i.e., missions have a valid number of agents committed to them), the agents can pursue

the goals in the order defined by the plan. The goals are distributed among the agents according to the missions to which they have committed and the norms defined in the normative specification.

4. The social scheme is finished when the root goal of the scheme either is achieved or was deemed impossible to achieve.
5. When no agent is committed to the missions of the social scheme, the social scheme entity can be detached from its corresponding group entities.
6. Once detached, the social scheme entity can be deleted by the agents.

Note that various social scheme entities may be created at any time from the same social scheme specification.

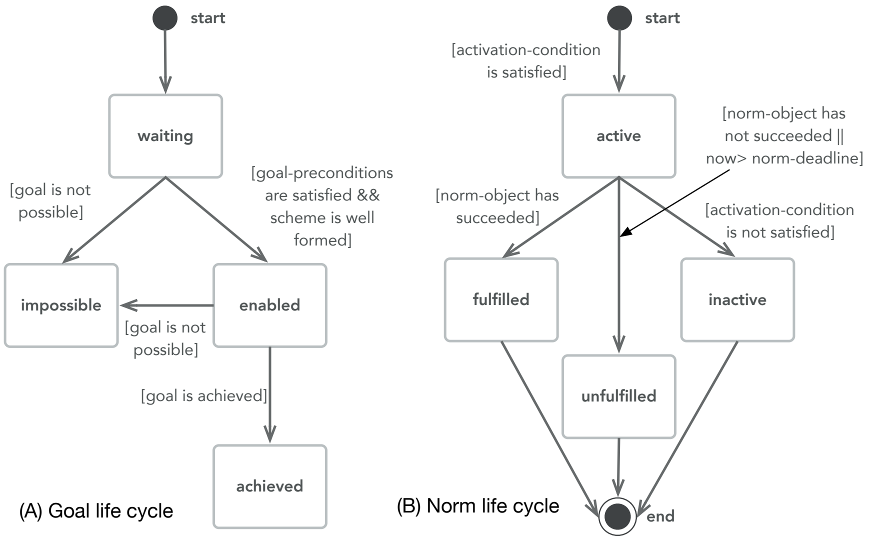

start
waiting
[goal is not possible]
[goal-preconditions are satisfied && scheme is well formed]
impossible
enabled
[goal is not possible]
[goal is achieved]
achieved
(A) Goal life cycle
start
[activation-condition is satisfied]
active
[norm-object has not succeeded || now> norm-deadline]
[norm-object has succeeded]
[activation-condition is not satisfied]
fulfilled
inactive
unfulfilled
(B) Norm life cycle
end

Figure 8.9
Life cycle of (A) a goal in a scheme entity and (B) a norm in a normative entity.

With regard to the *goal life cycle*, during the execution of a social scheme the goals appearing in the plan can be in one of the following states (see figure 8.9A): *organizational goal life cycle*

- *waiting* The goal cannot be pursued yet because it depends on the satisfaction of other goals (called goal preconditions) or on the well-formedness of the social scheme entity. The set of goal preconditions is deduced from the plan of the social scheme, that is, from the operators and from the dependencies between goals. This state is the initial state of every goal.

- *enabled* The goal can be pursued once the social scheme entity is well formed and the goal preconditions have been satisfied. Agents committed to a mission containing enabled goals can pursue them.
- *achieved* Agents committed to the goal have been able to achieve it.
- *impossible* The agents committed to the goal concluded that they will not be able to achieve it (goal is not possible, as shown in figure 8.9).

Note that the change of the state *waiting* to *enabled* is performed by the social scheme entity, whereas the change from state *enabled* to *achieved* is caused by the agents' behavior.

normative entity life cycle

Normative entity life cycle The *life cycle of a normative entity* has the following steps:

- Creation of a normative entity and connection to its corresponding group entity and social scheme entity is done in accordance with the normative specification.
- Once the social scheme entity is under the responsibility of its group entity, the normative state is composed of *mission* norms. These normative expressions are built from the abstract norms of the normative entity, replacing roles by the agents that play them in the group entity. Once created, these norms follow the norm life cycle presented below.
- The well-formedness of the social scheme entity is checked, that is to say that all mission norms are fulfilled (agents are committed to the missions referred to in the norms) and all constraints of the social scheme are satisfied. Once well formed, the normative entity is updated with *goal* norms as soon as the corresponding goal status becomes enabled according to its setting in the corresponding plan managed by the social scheme entity. Note that goal norms are obligation expressions that can be fulfilled or unfulfilled given the status of the corresponding goal (achieved or not). They are created from the mission norms as follows: obligation on missions involves obligation on goals, and permission on missions involves obligation on goals as soon as committed to the missions. We should note that commitment to a permitted mission implies creation of obligation on the goals belonging to it, because we consider that once committed to the mission, the agent has to achieve the goals in order to not create inconsistent behavior.
- Once all norms have been fulfilled, then the normative entities are detached from their social scheme and group entities. An agent can remove its commitments only if its obligations have already been fulfilled. If that is not the case (i.e., an agent has not fulfilled its obligations), the system does not allow it to be relieved from its commitments to missions.

An instantiated norm (both mission and goal norms) may have the following status (see figure 8.9B)

- *active* The activation condition of the normative expression holds (in the case of a mission norm, this is the activation condition defined in the abstract norm, whereas in the case of a goal norm, this is the fact that the goal has become enabled).
- *fulfilled* The object of the deontic modality of the norm has succeeded (a commitment for a mission norm, or goal achievement for a goal norm).
- *unfulfilled* In the case of obligation, the object of the deontic modality has not been successfully concluded (with a commitment for mission norms, or achievement for goal norms) or the deadline has passed. The norm has been violated.
- *inactive* The activation condition no longer holds.

Agent participating in an organization entity To conclude the presentation of the organization execution model, we present the *life cycle of a role-player agent*; that is to say, an agent that plays a role in the organization entity:

role-player agent life cycle

1. The agent adopts roles in groups, becoming a role player in the corresponding group entities. The adoption of a role by an agent is constrained by the role-cardinality and role-compatibility constraints. That is to say that an agent should have a clear strategy for adopting roles in an organization.
2. The agent undertakes the commitments to missions in social scheme entities. The commitment to a mission is constrained by the normative specification as well as the constraints on missions.
3. The agent undertakes the goals associated with the committed missions in accordance with the progress of the scheme execution in which the goals appear (see figure 8.10).
4. Removal of commitments. This removal is also constrained; an agent can remove its commitments only if its obligations were already fulfilled. If an agent does not fulfill its obligations, the system should not allow it to remove the commitments.
5. The agent may decide to leave its roles if it has no outstanding commitments and may quit the corresponding group entity.

```
// Domain level actions: satisfaction of goals
+!goal2[scheme(sch1)]  <- .println("satisfying goal2 in sch1");  +goal2.
+!goal3[scheme(sch1)]  <- .println("satisfying goal3 in sch1");  +goal3.
+!goal4[scheme(sch1)]  <- .println("satisfying goal4 in sch1");  +goal4.
+!goal2[scheme(ssch1)] <- .println("satisfying goal2 in ssch1"); +goal2.
+!goal3[scheme(ssch1)] <- .println("satisfying goal3 in ssch1"); +goal3.
+!goal4[scheme(ssch1)] <- .println("satisfying goal4 in ssch1"); +goal4.
```

Figure 8.10
Agent plans to achieve organizational goals.

From this presentation, we can note that whereas the changes of the group and social scheme entities are the direct results of the agents' actions—for instance creating groups, adopting roles, and committing to missions—the normative state observes its monitored group and social scheme entities to activate/deactivate norms and trigger events that will be perceived by the agents. The regulation specified by the norms depends only on how the structure and coordination evolve in the system under the actions of the agents. Agents thus have no action to change the normative entity directly, whereas they have to act on the group and social scheme entities.

Situating Organization in an Environment

In the type of complex systems targeted with the MAOP approach, it is important to consider the interpretation of norms with respect to the environment in which the agents are situated. This interpretation is usually referred in the literature as *constitution* (Searle 1997, 2010; Balke et al. 2013).

For instance, in an auction scenario, norms may regulate payments and bids performed by the agents participating to the auction. How payments are realized (using paper bills or any other means) and how bids are realized (e.g., by raising hands or shouting) are not explicitly defined by the norms. However, these definitions are of first importance in order to monitor the activation, violation, and fulfillment of norms. We need to state what in the environment *constitutes* payments and bids for the norms. For instance, it is necessary to specify that the raising of hands is an event in the environment that *counts as* a bid in the auction process.

Constitution raises two issues for MAOP. First is how to define the relation between concepts appearing in the organization dimension and their counterpart in the environment dimension (e.g., interpreting the execution of some operation on an artifact as the achievement of an organizational goal and recognizing the execution of a set of operations on artifacts as the fact that some agent is playing some role). Second is how to define it so that the same set of operations on artifacts can be referred to by different norms and reciprocally to have the possibility of using the same norm with, for instance, different sets of operations counting as the norm fulfillment (e.g., using either the raising of hands or blinking of eyes to count as realizing a bid).

The literature has proposed different models (Broersen et al. 2013; Broersen and van der Torre 2012; Boella and van der Torre 2004) that introduce an intermediate entity between the environment and the normative entities, called a *constitutive entity*, in which constitutive statements are placed.

It is also proposed to introduce explicit representations of constitutive rules to capture the interpretation of the agents, events, and states of the environment as *institutional facts*, which helps the management of the norms. Although the literature proposes specialized approaches for norms and for constitutive rules, they are not usually connected to each other (Boella and van der Torre 2006). The *situated artificial institution* model introduced in (de Brito et al. 2018, 2017) proposes solutions and architectures addressing these questions and is integrated into JaCaMo.

Conversely to this interpretation of the environment as a provider of concrete facts for norms monitoring, organization may *empower* elements of the environment by allowing them to control and regulate actions or perception of the agents. As shown in (Okuyama et al. 2013) and (Piunti et al. 2009), this dynamic relation is a practical way of situating organizations in an environment.

8.4 Bibliographical Notes

The concept of organization and its related concepts have started to be considered an important (independent) dimension for multi-agent systems by seminal work in the 1980s (Fox 1981; Corkill and Lesser 1983; Pattison et al. 1987; Gasser et al. 1989). Given its multidisciplinary nature, the organization dimension was approached from different perspectives, from sociology (Ferber and Gutknecht 1998; Demazeau and Rocha Costa 1996; Bond 1990) to ethology (Drogoul et al. 1995), resulting in several organizational models (e.g., AGR [Ferber and Gutknecht], TeamCore [Tambe], Islander [Esteva et al.], and Moise [Hübner et al.]). Moise is the model on which the organization dimension described in this book is grounded. It originated from the refactoring of the initial version presented in (Hannoun et al. 2000). Extensive comparisons of those different models have been proposed in various journal papers and books (Coutinho et al. 2009; Ossowski 2012; Dignum 2009; Aldewereld et al. 2016).

The initial proposals to program that dimension were then proposed in platforms such as MadKit (Gutknecht and Ferber 2000), Karma (Pynadath and Tambe 2003), Ameli (Esteva et al. 2004), ORA4MAS (Hübner et al. 2010), OMNI (Dignum et al. 2004), and 2OPL (Dastani et al. 2009). For a detailed account of the area, the International Workshop on Coordination, Organizations, Institutions and Norms (COIN) has published a series of books (e.g., Ghose et al. 2015).

In this chapter and in the book, we did not consider an aspect of organization in multi-agent system that is called *emergent organization*. An emergent organization is an organization entity for which the properties are unknown at the level of the agents themselves, that is to say that there does not exist any representation in the agents or outside of the agents. Stakeholders may observe an organization entity that is a side effect of the interactions taking place between the agents in the MAS. This is usually called an organization in complex systems (Morin 1977) or self-* domains.[2]

Exercises

Exercise 8.1 Define the organization specification for the management of the collaborative writing of a journal paper. The management of the writing is handled by assistant agents that support the users in the execution of the global workflow for executing this process. We will consider that the structure of this organization has only one group to make collaborate an editor (not more than one) and a set of writers (to keep the writing manageable, we allow one to five writers to participate in this group). The editor has authority over writers. It is possible to be editor and writer at the same time. To write the submitted version of the paper under this structure, a draft version of the paper has to be produced first, followed by the submitted version. The draft version is composed of a title, an abstract, an introduction, and the list of section names. Whereas the draft version is produced only by the editor, the filling in of the sections for the submitted version is done by the writers except for the conclusion, which is written by the editor. A set of references provided by each of the writers is added to the submitted version.

Exercise 8.2 Define the organization specification for the management of a master training program. The management of the administrative tasks is realized by a set of assistant agents who help the educational staff and the students in their procedures. Because it is a complex process, we focus here on the *registration for the courses* process: after being registered as a master's student in the master's program, each student must request his/her registration for the courses he/she is interested in. This requires the student to fill out a form, to have the agreement of the professor responsible for the course, and then to give it to the secretary of the administration who asks the teaching director to validate it and then, if it is validated, requests the database manager to store it. The student is then informed of the success/failure of his/her application.

Exercise 8.3 Reconsider the domestic robot exercise presented in chapter 6 and extend it by considering that several owners, each assisted by a robot, have access to a unique fridge and to one supermarket. In the case of a lack of beers in the fridge, only one robot can order more beer using the supermarket's home delivery service. Instead of being hardwired into the robot by the Department of Health, the application defines several policies depending on the age of the owner. When launching the system, the proper set of policies is created according to the age of the owner.

2. The series of conferences on Self-Adaptive and Self-Organizing Systems, http://www.saso-conference.org/, can be consulted for more information.

9 Programming Organizations of Situated Agents

In this chapter we see the organization dimension in practice in a further extension of the smart room scenario, involving more articulated and complex coordination patterns among the agents. We redesign the scenario on the basis of the organization concepts and programming abstractions introduced in the previous chapter, discussing in particular how coordination patterns among the agents can be reframed to achieve more flexibility and openness. This chapter also explores how agents can be programmed to reason about the organization in which they are participating so that, for instance, they can discover some role of interest in terms of missions, goals, and duties.

9.1 Programming an Organized Smart Room

In this chapter, we extend the previous example of managing a single room temperature by introducing a set of rooms with different policies for managing the temperature in each of the rooms (e.g., a voting mechanism in one room and a first-come-first-served policy in another room). The organization will help us to define these policies in an abstract way by using a suitable language. Thus when we want to apply different policies in the rooms, we, or the agents themselves, can simply select a different organization. For instance, in the single room temperature application of the chapter 7, the voting process is hard-coded within the `room_controller` or `personal_assistant` agents and `VotingMachine` artifact programs. However, agent and artifact programming languages are not designed to easily manage such a process. The aim here is to turn the voting process explicitly programmed in the organization dimension into a language designed for that, so that we (the software developers or the agents themselves) can define and change the process to modify the temperature by changing only the organization without touching the agent or artifact programs.

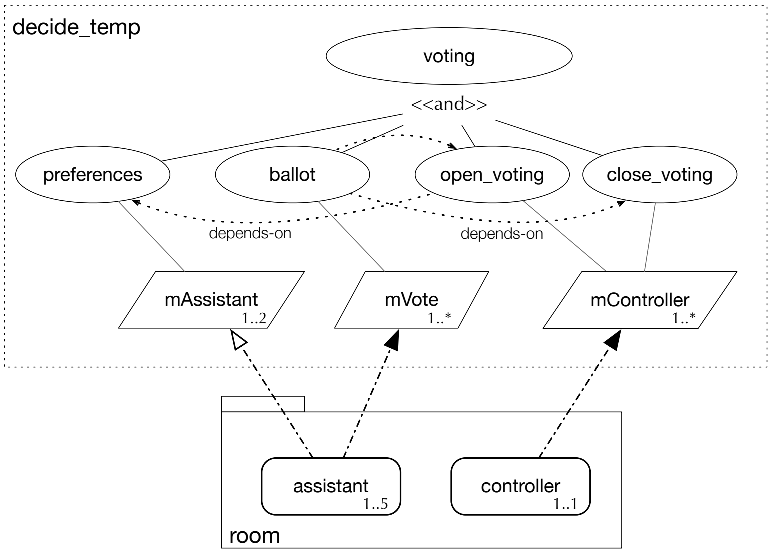


Figure 9.1
Organization specification for the organized smart room.

Defining the organization for the smart room We start by specifying an organization on top of the voting process we used in the previous chapters. As described in the chapter 8, the specification of the organization has three parts: structural (groups, roles, ...), functional (social schemes, goals, ...), and normative (norms). In the structural part of this organization, we have the following roles to be played by agents:

defining the structural specification

- `assistant` This role is played by agents who want to keep the room temperature according to the preferences of their owners; and
- `controller` This role is played by agents who are able to handle the `hvac`.

The group, identified by `room` in figure 9.1, should have exactly one agent playing the role `controller` and up to five agents as `assistant`. The *cardinality* ($< min, max >$) of roles `controller` and `assistant` are $< 1, 1 >$ and $< 1, 5 >$ respectively. Group entities with this configuration of agents are considered *well formed*. By defining this group, we are constraining the participation of agents in the temperature choice to those who joined the group by playing one of the two roles as well as constraining the number of such agents. The organization also defines its own goals as well as when and by which agents they have to be achieved—the functional part of the specification. For the voting protocol, we can consider the following goals:

defining the functional specification

- `preferences` Collect temperature preferences from users;
- `open_voting` Start the voting processes;
- `ballot` Agents vote for their preferred options; and
- `close_voting` Stop the voting processes and determine the room temperature.

These goals are subgoals of the `voting` goal and part of the scheme `decide_temp` as illustrated in figure 9.1. The achievement of the subgoals implies the achievement of the goal `voting`. The scheme also states that these goals should be achieved exactly in the order they are presented here.

defining the normative specification

We finish the organization specification with some norms. They declare the set of duties and permissions that agents will have to fulfill while they play roles in the `room` group. The norms are the following:

- Agents playing `assistant` are permitted to achieve the goal `preferences`;
- Agents playing `assistant` are obliged to achieve the goal `ballot`; and
- Agents playing `controller` are obliged to achieve the goals `open_voting` and `close_voting`.

An obligation implies a permission, so `personal_assistant` agents are also permitted to achieve that goal and no other role has that permission. With the normative specification, we are constraining who can achieve the organizational goals: they can be achieved only by agents that are obliged or permitted to do so through the roles they are playing.[1]

defining the organization specification

The group, scheme, and norms described so far constitute a *type* of organization, and its specification is coded in an XML file—it is called *Organization Specification* (OS). This file has three parts corresponding to the three specifications that we just defined: structural, functional, and normative. For the organized smart room example, the content of the file is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="http://moise.sf.net/xml/os.xsl"
                 type="text/xsl" ?>
<organisational-specification id="room_org"

  os-version="0.11"
  xmlns='http://moise.sourceforge.net/os'
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xsi:schemaLocation='http://moise.sourceforge.net/os
                      http://moise.sourceforge.net/xml/os.xsd' >

```

1. There are implicit prohibitions in the set of norms of this example, for instance, agents not playing `assistant` role are prohibited from achieving the goal `ballot`.

```
  <structural-specification>
        <group-specification id="room" >
            <roles>
                <role id="assistant"  min="1" max="5" />
                <role id="controller" min="1" max="1" />
            </roles>
        </group-specification>
  </structural-specification>

  <functional-specification>
        <scheme id="decide_temp" >
            <goal id="voting">
                <plan operator="sequence" >
                    <goal id="preferences"  ttf="5 seconds" />
                    <goal id="open_voting"/>
                    <goal id="ballot" ttf="10 seconds">
                        <argument id="voting_machine_id" />
                    </goal>
                    <goal id="close_voting" />
                </plan>
            </goal>

            <mission id="mAssistant" min="1" >
                <goal id="preferences"/>
            </mission>

            <mission id="mVote" min="1" >
                <goal id="ballot"/>
            </mission>

            <mission id="mController" min="1" >
                <goal id="open_voting"/>
                <goal id="close_voting"/>
            </mission>
        </scheme>

  </functional-specification>

  <normative-specification>
    <norm id="n1a"          type="permission"
          role="assistant"  mission="mAssistant" />
    <norm id="n1b"          type="obligation"
          role="assistant"  mission="mVote" />
    <norm id="n2"           type="obligation"
          role="controller" mission="mController" />
  </normative-specification>
</organisational-specification>
```

We can note that this file has more details than the specification shown in figure 9.1. The figure represents only part of the specification, so more details are added in the OS file.

- Some goals have a `ttf` (time to fulfill) attribute (an example is shown in line 25). The value of this attribute defines the deadline for the agent to fulfill the related obligation. For example, once the norm that obliges some agent to achieve `preferences` is activated, the agent has five seconds to achieve it.
- The goal `ballot` has an argument (`voting_machine_id` in line 28). The value of this attribute is defined at runtime and will be known by the agents participating in the scheme. In this case, they use the value to select the voting machine with which they are obliged to vote.
- The missions (`mAssistant`, `mVote`, and `mController`) are specified to assemble related goals (lines 34...). Missions define the set of organizational goals to be achieved by an agent, and thus, thanks to the social plan that structures the set of goals, an agent can know when it has to take part in the scheme execution. In this organization, the social scheme requires agents in three missions, at least one agent in each. Only when enough agents have committed to the missions is the scheme *well formed* and can start its execution.
- Norms are defined for missions and not for goals (lines 50...). Agents are obliged or permitted to commit to a mission of a social scheme. Once committed, they are always obliged to achieve the goals that are part of those missions.

Deploying the organized smart room This organization specification can be instantiated, creating an *Organization Entity* (OE), either by an entry in the main application file (`.jcm` file) or by the agents themselves. Considering the application file option, the following excerpt of code illustrates the creation of an organization entity called `smart_house_org` based on the specification introduced previously (which is in a file named `smart_house.xml`).

deploying the organization entity in the application file

```
mas org_voting {
    // ... agents and workspaces are created here
    organisation smart_house_org : smart_house.xml {
            group r1 : room {
                players: pa1 assistant
                         pa2 assistant
                         pa3 assistant
                         rc  controller
                responsible-for : temp_r1
            }
            scheme temp_r1: decide_temp
    }
```

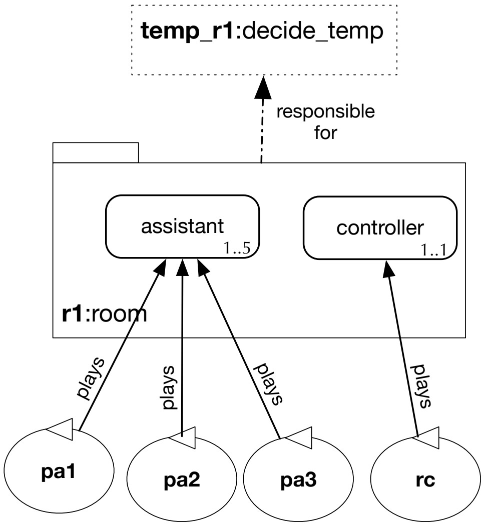


Figure 9.2
Organization entity and group entity `r1`.

In this organization entity, based on the group specification `room`, we create one *group entity*, identified by `r1` (line 4), in which agents play some roles as listed after the `players` keyword (see figure 9.2). In this case, we are assigning roles for the agents in the application file. Of course, as for many initial definitions written in this file, the role adoption could be coded inside the agents, and we discuss how to do so subsequently. The organization also has one *social scheme entity*, identified by `temp_r1f` (line 11), and agents in group entity `r1` will be responsible for its missions and goals (line 9).

Programming the agents to act in the organized smart room We now focus on the agent dimension and how to program the agents that participate in this organization. An important principle in the MAOP approach is that agents may achieve organizational goals in their own way. We have thus to program plans in the agents to achieve organizational goals; for example, the `room_controller` agent should have plans for the organizational goals `open_voting` and `close_voting` that it could have to achieve when participating in the organization:

```
+!open_voting[scheme(S)]
   <- // get temperature preferentes sent by message from assistants
      .findall(T,pref_temp(T)[source(_)],Options);

      // get voters from organization (agents playing assistant)
      .findall(A, play(A,assistant,_), Voters);
```

```
7
8          // open "voting machine" for votes
9          vm::open(Options, Voters, 4000);
10         .print("Options are ",Options," voters are ",Voters);
11
12         // set the argument of organizational goal "vote"
13         setArgumentValue(ballot,voting_machine_id,v1)[artifact_name(S)];
14      .
15
16  +!close_voting
17      <- ?vm::result(T);
18         .println("Creating a new goal to set temperature to ",T);
19         .drop_desire(temperature(_));
20         !!temperature(T);
21      .
```

The `VotingMachine` artifact is still used (as illustrated in figure 7.3) because it offers the proper usage interface to implement the voting process actions. The `room_controller` agent uses this artifact to achieve the `open_voting` goal. We highlight line 6 where a consult of belief `play(Ag,Role,Group)` is used to identify all agents playing role `assistant`. Broadcast messages between the agents are no longer necessary to find out who are the assistants. The `play/3` belief in the agent mind comes from the *organization entity*. The management of this organization entity as defined in the organization execution model described in chapter 8 is implemented by dedicated artifacts, called *organizational artifacts*. They offer a set of observable properties that provides to the agents a view on the state of the organization entity, as well as a set of operations for the agents to change the organization. Any agent focusing on them can perceive and change the organization state.

The program for `personal_assistant` agents has also plans for handling the organizational goals `preferences` and `ballot`:

```
+!preferences
  <- ?pref_temp(T); // consults my personal preference
     .send(rc, tell, pref_temp(T)); // and send to rc
  .

+!ballot
  <- .wait(300); // thinking on how to vote...
     // get the name of the voting machine artifact
     // defined for organizational goal "vote"
     ?goalArgument(_, ballot, "voting_machine_id", VMName);
     // get the workspace id where the voting machine is
     ?joined(vmws,VWId);
     lookupArtifact(VMName,VMId)[wid(VWId)];
     // focus on the voting VotingMachine using namespace vm
     vm::focus(VMId)[wid(VWId)];

     // consult the temperature options
```

```
      ?vm::options([First,Second|_]);
      // simply vote on the second option (!)
      vm::vote([Second]);
   .

{ include("$moiseJar/asl/org-obedient.asl") }
```

Initially, we consider fully obedient agents—that is, they obey obligations, permissions, and prohibitions as determined by the norms of the organization. The plans written in the file included by the `include` command in the last line of the preceding program gives this behavior to the assistant agents. More precisely, including file `org-obedient.asl` adds the following plans into the agent's plan library:

obedient agent

```
+obligation(Ag, MCond, committed(Ag,Mission,Scheme), Deadline)
    : .my_name(Ag)
   <- commitMission(Mission)[artifact_name(Scheme)].
+obligation(Ag, MCond, done(Scheme,Goal,Ag), Deadline)
    : .my_name(Ag)
   <- !Goal[scheme(Scheme)].
```

We can notice that these plans react to the addition of beliefs:

```
obligation(Ag, MCond, What, Deadline)
```

As the belief `play/3`, the belief `obligation/4` comes from the organization, and all agents focusing on organizational artifacts will perceive them. The obligation can be read as "agent `Ag` is obliged to `What` before `Deadline` while `MCond`." The `Deadline` argument is the result of adding the "time to fulfill" (`ttf` as defined in the OS) to the time when the norm is activated. If `ttf` is not provided in the OS, there is no deadline. The `What` argument is the organizational fact the agent is obliged to bring about. As explained in chapter 8, corresponding respectively to the mission and goal norms, two kinds of `What` arguments are considered:

- `committed(Ag,Mission,Scheme)` Agent `Ag` is obliged to commit to `Mission`. This kind of obligation is created when the agent plays a role that is obliged to `Mission` (as specified in the OS norms) in a group that is responsible for `Scheme`. Fulfilling this obligation is simple: the agent only has to commit to the mission using the operation `commitMission` that is available in organizational artifacts.
- `done(Scheme,Goal,Ag)` Agent `Ag` is obliged to *achieve* `Goal`. This kind of obligation is created when the goal `Goal` is *enabled*. An organizational goal is enabled when (1) its scheme entity is well formed, and (2) its precondition goals are achieved. In our example, the achievement of goal `open_voting` is the precondition of `ballot`, as implied by the decomposition of goal `voting` in the

social scheme. The plan in lines 4–6 reacts to this obligation by including the organizational goal as an agent goal (line 6). For now, the agent fulfills this obligation by completing the execution of a plan to achieve `Goal`.[2] Even though an agent may have fulfilled an obligation by achieving `Goal`, in this example, a goal is considered achieved only when all committed agents have done so.

Similar to the creation of `obligation` from the norms, `permission` is also created, with the same arguments as obligations.

The `personal_assistant` agents have a plan for them because, regarding norm `n1a` of the OS, they are permitted (rather than obliged) to commit to mission `mAssistant`:

```
+permission(Ag, MCond, committed(Ag,Mission,Scheme), Deadline)
    : .my_name(Ag)
   <- commitMission(Mission).
```

Besides plans to react to changes issued in the organization entity (obligations and permissions) and those to achieve organizational goals issued from the norms, a `personal_assistant` agent continues to react to changes in its environment. The following plan reacts to new activities of the user in the room:

```
+activity(A) : A \== "none"  <- resetGoal(voting).
```

This plan uses the `resetGoal` action to restart the scheme execution. This operation sets `Goal` and all that follows it in the scheme entity as not achieved. For example, by resetting goal `voting`, all its subgoals (`preferences`, `open_voting`, ...) and itself are considered unachieved; by resetting goal `ballot`, the goals `ballot` and `close_voting` are not considered satisfied anymore. By resetting the root goal `voting` (as done in the above plan), the `personal_assistant` agent effectively starts the scheme again, which will trigger new obligations for agents committed to the missions including those goals.

Organization Management in JaCaMo

The management of an organization entity is realized by a set of dedicated *organizational artifacts* that the agents can focus on and act on when playing

2. This kind of fulfillment is far from ideal, because it depends on the internal state of the agent. It is based on the execution of an agent's plan, about which the designer may not even know in the case of open systems, and so its execution could mean nothing regarding the obligation. Instead, the fulfillment should be based on concrete changes in the environment. For instance, the door should be perceived to be open to fulfill the obligation `done(s1,open(door),bob)`; simply running a plan to open the door is not enough because the plan may fail. A better approach for the fulfillment of obligation is proposed by de Brito et al. (2018), based on the notion of situated artificial institutions (SAI) (see the research corner on page 140).

a role in this organization entity. By focusing on these artifacts, an agent can obtain the following beliefs on the state of the organization entity (group/-social scheme/normative entities):

`specification(S)[artifact_name(_,A)]` `S` is the specification of `A` (`A` being a group, scheme, or organization identifier).

`play(A,R,G)` Agent `A` is playing role `R` in group `G`.

`schemes(L)[artifact_name(_,G)]` Group `G` is responsible for the schemes belonging to the list `L`.

`commitment(A,M,S)` Agent `A` is committed to mission `M` in scheme `S`.

`groups(L)[artifact_name(_,S)]` The groups in the list `L` are responsible for scheme `S`.

`formationStatus(S)[artifact_name(_,A)]` The formation status for scheme or group `A` is `S` (possible values for `S` are `ok` and `nok`).

`goalState(S, G, LC, LA, T)` Goal `G`, of scheme `S`, is in state `T` (possible values for `T` are `waiting`, `enabled`, and `achieved`);
`LC` is the list of agents committed to the goal, and `LA` is the list of agents that have already achieved the goal.

`goalArgument(S,G,A,V)` Argument `A` of goal `G` has value `V` in scheme `S`.

`obligation(A,R,G,D)` Agent `A` is obliged to achieve `G` before `D` while `R` holds.

`permission(A,R,G,D)` The agent `A` is permitted to achieve `G` before `D` while `R` holds.

Besides beliefs, the following events can also be perceived by the agent from the signals produced during the organization life cycle:

`oblCreated(O)` Obligation `O` was created.

`oblFulfilled(O)` Obligation `O` was fulfilled.

`oblUnfulfilled(O)` Obligation `O` is unfulfilled.

`oblInactive(O)` Obligation `O` is inactive.

`normFailure(F)` There was a failure `F` in the normative system.

The following actions are provided to the agents by the organizational artifacts to change the organization entity state:

`createGroup(Name,Type,ArtId)[artifact_name(O)]` Creates a new group of name `Name`, following the group specification `Type` defined in the OS used to create organization (O). The organizational artifact (`GroupBoard`) that manages it is identified by `ArtId`.

createScheme(Name,Type,ArtId)[artifact_name(O)] Creates a new scheme of name `Name`, following the scheme specification `Type` defined in the OS used to create organization (O). The organizational artifact (`SchemeBoard`) that manages it is identified by `ArtId`.

adoptRole(R)[artifact_name(G)] Adopt role `R` in group `G`.

leaveRole(R)[artifact_name(G)] Leave role `R` in group `G`.

addScheme(S)[artifact_name(G)] Add scheme `S` to the responsibility of group `G`.

removeScheme(S)[artifact_name(G)] Remove scheme `S` of the responsibility of group `G`.

commitMission(M)[artifact_name(S)] Commit to mission `M` in scheme `S`.

leaveMission(M)[artifact_name(S)] Leave mission `M` in scheme `S`.

resetGoal(G)[artifact_name(S)] Set goal `G` as not satisfied in scheme `S`.

setArgumentValue(G,A,V)[artifact_name(S)] Set `V` as the value of argument `A` of goal `G` in scheme `S`.

Executing the agents in the organized smart room As explained in chapter 8, participating in the organization means that the agents adopt roles and commit to missions to achieve goals under their duties as stated by the norms.

The *organizational artifacts* building the *organization management infrastructure* in JaCaMo interpret the organization specifications in order to coordinate the agents' activities at runtime. JaCaMo starts the coordinated execution of a created social scheme entity as soon as the group entity that is responsible for it is *well formed* (that is, at least the minimum number of agents that are required to play the roles have already adopted those roles). In our example, because the group entity `r1` is responsible for the social scheme entity `temp_r1`, as soon as `r1` is well formed, the following obligations and permissions are created by the organization:

managing the organization with organizational artifacts

```
permission(pa1, n1a, committed(pa1, mAssistant, temp_r1), _).
permission(pa2, n1a, committed(pa2, mAssistant, temp_r1), _).
permission(pa3, n1a, committed(pa3, mAssistant, temp_r1), _).

obligation(pa1, n1b,  committed(pa1, mVote, temp_r1), _).
obligation(pa2, n1b,  committed(pa3, mVote, temp_r1), _).
obligation(pa3, n1b,  committed(pa3, mVote, temp_r1), _).
obligation(rc,  n2,   committed(rc, mController, temp_r1), _).
```

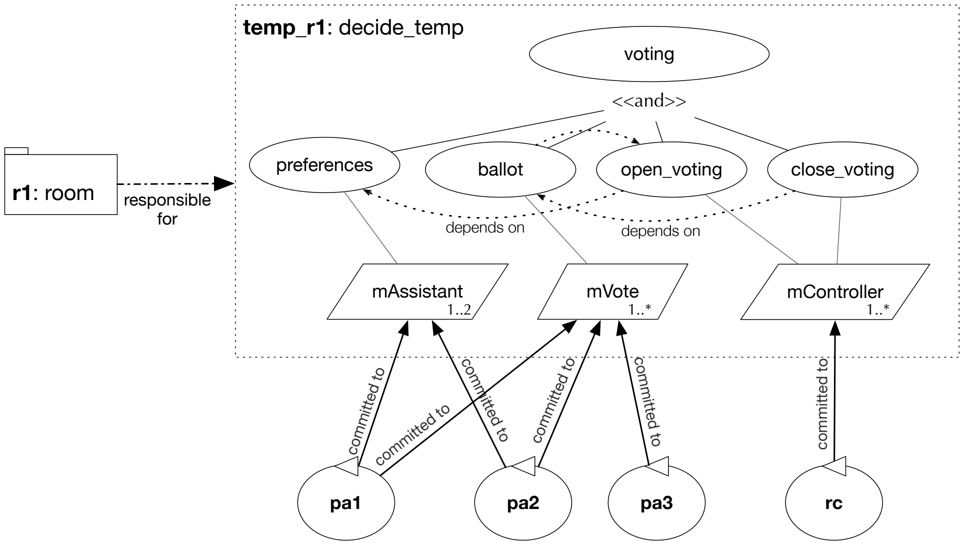


Figure 9.3
Organization entity and social scheme entity `temp_r1`.

Having programmed obedient agents, the `personal_assistant` and `room_controller` agents behave as ordered by the organization and will commit to their missions. The organization entity state after the commitments is as shown in figure 9.3. With these commitments, the social scheme entity becomes also *well formed* (which in this case means that all agents have committed to performing the corresponding goals under their missions). Because it is well formed, JaCaMo determines new obligations for the goals of the social scheme.[3] Initially, because the organizational goal `preferences` has no precondition, it is enabled and then three obligations are created:

```
obligation(pa1,goaln, done(temp_r1,preferences,pa1), "2018-9-5 8:47:57").
obligation(pa2,goaln, done(temp_r1,preferences,pa2), "2018-9-5 8:47:57").
obligation(pa3,goaln, done(temp_r1,preferences,pa3), "2018-9-5 8:47:57").
```

Again, obedient agents will execute their plans to achieve the organizational goal `preferences`. In our example, their plans consist of simply sending a message to the `room_controller` agent with the user-preferred temperature (cf. section 9.1). When all agents committed to `preferences` have achieved their goals (in our

3. The OS file contains explicit norms linking roles to missions of a particular application. Besides those, generic norms are included in all applications. One generic norm is "When a goal *g* of mission *m* is enabled, agents committed to *m* are obliged to *g* before *ttf*."

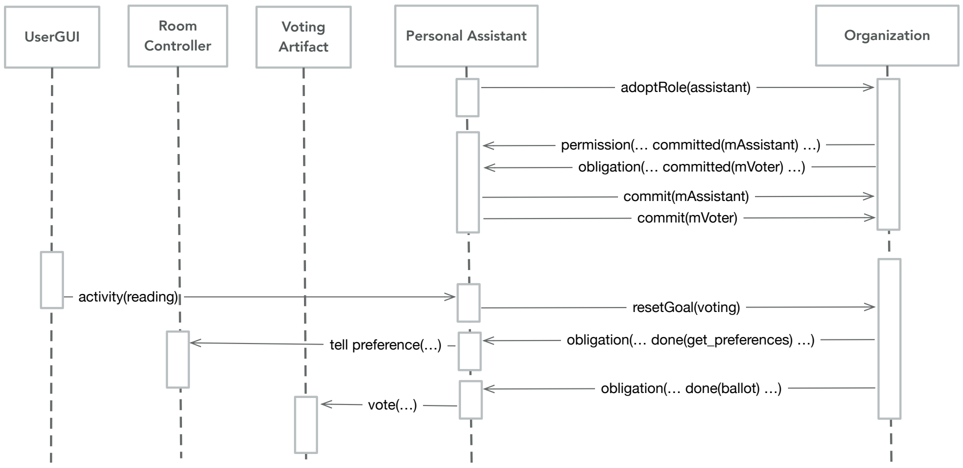


Figure 9.4
Interaction of the `personal_assistant` agent with the other elements of the system.

example, `pa1`, `pa2`, and `pa3`), the organizational goal is considered satisfied by the organization. The achievement of this goal enables the goal `open_voting`, creating then the following obligation for the `room_controller` agent:

```
obligation(rc, goalNorm, done(temp_r1,open_voting,rc), _).
```

The `room_controller` agent fulfills the obligation by opening the voting process in the `VotingMachine` artifact. This again enables the next goal, creating new obligations, and if fulfilled, enabling the next goal, and so on. Figure 9.4 illustrates the interaction between the agents, the artifacts, and the organization—for the sake of simplicity all organizational artifacts involved in the organization management are grouped in a sole component—from the perspective of a `personal_assistant` agent.

In the execution described so far, we can note that the JaCaMo organization management infrastructure computes itself when new goals have to be pursued because its dependent goals have been achieved by other agents. The enabling of goals activates some goal norms that create obligations for the agents committed to the corresponding missions. The (obedient) agents then can start acting to achieve that particular goal. The coordination is thus managed by the organization by issuing obligations for the agents. With the addition of an organization, we do not find code related to the coordination of the voting process either in the agent programs or in artifact programs: agents simply have plans for handling organizational goals.

9.2 Changing the Organization

We have seen how the implementation of the coordination was moved from the agent to the organization dimension. Although this action has required learning several new concepts to write new lines of code, we discuss in this section the interest of using such approach to program coordination instead of hard-coding it in the agents.

Suppose we want to change the way the temperature is defined in the organized smart room so that the `room_controller` agent collects preferences and sets the target temperature to be the average of the requested temperatures. We can do that by following a few steps:

1. Copy the previous OS to a new file called `smart_house_s.xml` and change the organizational goals of the social scheme as shown in figure 9.5:

```
<goal id="voting">
      <plan operator="sequence" >
            <goal id="preferences"  ttf="5 seconds" />
            <goal id="set_average" />
      </plan>
</goal>
```

2. Change the organization entity so that it uses the new OS:

```
organisation smart_house_org : smart_house_s.xml
```

3. Add a plan in the `room_controller` agent program to achieve the new organizational goal `set_average`:

```
+!set_average
   <- .findall(T,pref_temp(T)[source(_)],Options);
      .drop_desire(temperature(_));
      !!temperature( math.average(Options) ).
```

This plan retrieves all temperature preferences informed by the agents and then creates a new intention with the goal to set the target temperature to the average of all preferred temperatures (`!!temperature(...)`); before doing so, any currently ongoing attempts to set the target temperature must be dropped.

These changes are all that is needed—no changes in the voting artifact or participating agent programs. Moreover, we can come back to the voting process by simply selecting the previous OS. In that case, the plan for achieving the goal `set_average` will not be triggered. Following the same approach, different temperature selection schemes can be created by different organizations using the *same* program for the agents and artifacts.

We can now highlight some advantages of this approach:

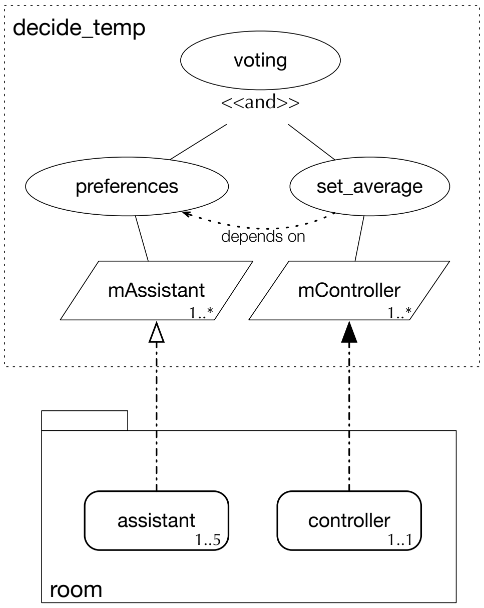


Figure 9.5
Organization specification to set the target temperature to the average preference.

- No exchange of messages is required between the agents for coordination purposes. The organization management infrastructure intermediates the coordination by interpreting and managing the organization entity life cycle.
- The order of the organizational goals is not implemented in the agents, and it is enforced by the organization. They have plans for organizational goals and that is all they need. To change the order of achievement of the goals, we simply change the social scheme specification. There is no need to change any agent code.
- In the case that we need to add more `personal_assistant` agents, we can simply assign them to play the `assistant` role and then they can participate in the voting processes, as long as they have plans for achieving the corresponding organizational goals. Playing this role is a necessary condition for that, and the right to participate comes with some duties. This is particularly useful in open systems where we do not know the behavior of incoming agents beforehand.
- The organization can be perceived by any agent, which can thus monitor the state of the organization (e.g., to discover which agents are not fulfilling their obligations).

9.3 Agents Deploying Their Organization

In the previous section, the organization is created by the application designer using the application file. The created agents then execute in the resulting organization entity. However, in some applications it is preferable that the agents themselves create the organization entity. The JaCaMo organization management infrastructure offers some actions for the agents to take control of their organization.

Considering the same organized smart room scenario, we can change it so that the `room_controller` agent creates the organization for its room, and it does so by selecting the best organization specification for the temperature control policy to be used in the room. To implement this version, we have to remove all `organization` code from the application file and add plans in the agents to build up the organizational entity.

The new `room_controller` agent performs the following steps:

1. Creates the organization entity where the group and social scheme entities will be deployed (lines 6–14);
2. Creates a group entity (lines 15, 16—based on the specification `room` defined in the `smart_house.xml` file);
3. Adopts the role `controller` in this group (line 19);
4. Announces the existence of the `room` group to the participants so that they can join in (line 22);
5. Waits for the participants to adopt their roles (line 25);
6. Creates the social scheme entity (line 28—based on the specification `decide_temp` defined in `smart_house.xml`); and
7. Sets the group entity to be responsible for this social scheme entity (line 29).

The code that implements these steps in the `room_controller` agent is as follows:

```
3   !create_org. // Initial goal
4
5   +!create_org
6      <- createWorkspace(shouseo);
7         joinWorkspace(shouseo, WspId);
8
9         // creates the organization entity
10        makeArtifact(shouseo,
11            "ora4mas.nopl.OrgBoard",
12            ["src/org/smart_house.xml"],
13            OrgArtId)[wid(WspId)];
14        focus(OrgArtId)[wid(WspId)];
15        createGroup(r1, room, GrArtId)[artifact_id(OrgArtId)];
16        focus(GrArtId)[wid(WspId)];
17
18        // adopts the role controller in the group
```

```
19        adoptRole(controller)[artifact_id(GrArtId)];
20
21        // announces to others the new group, so they can join
22        .broadcast(tell,new_gr(shouseo,r1));
23
24        // waits for the group to be well formed
25        .wait(formationStatus(ok)[artifact_id(GrArtId)]);
26
27        // creates the scheme
28        createScheme(temp_r1, decide_temp, SchArtId)[artifact_id(OrgArtId)];
29        addScheme(temp_r1)[artifact_id(GrArtId)];
30        focus(SchArtId)[wid(WspId)];
31     .
```

On the `personal_assistant` agents side, when they realize that a new group was created, they adopt the role `assistant` in it. The following code implements this behavior:

```
+new_gr(Workspace,GroupName)
   <-  joinWorkspace(Workspace,WspId);
       lookupArtifact(GroupName,GrArtId)[wid(WspId)];
       adoptRole(assistant)[artifact_id(GrArtId)];
       focus(GrArtId)[wid(WspId)];
   .
```

Because this code creates the same organization as in the previous section, nothing else needs to be changed and the application will run as before. However, with dynamic creation of the organization, we can program better solutions for our organized smart room application. For instance, agents can exploit this feature to instantiate a *new* scheme entity each time a new target temperature should be selected (instead of using `resetGoal`). To implement this new solution, we have to remove lines 24–30 of the `room_controller` agent, because the social schemes entities will be created on demand, and add the following lines in the code of `personal_assistant` agents:

```
+activity(A) : A \== "none"
   <- .print("New user activity ",A);
      // gets some art ids
      ?focused(shouseo,r1,GrBoardId);
      ?joined(shouseo,WspId);

      // computes a new schme id
      ?schemes(L); // L is the list of group's schemes
      .concat("sch_", .length(L)+1, Name);

      // creates the new scheme
      createScheme(Name, decide_temp, SchArtId)
                    [wid(WspId),artifact_name(shouseo)];
      addScheme(Name)[artifact_id(GrBoardId)];
      focus(SchArtId)[wid(WspId)];
   .
```

Reading the new code of the agents, we can see that some artifact operations like `makeArtifact` and `focus` are used to create and manage the organization entity. Agents access their organization entity through artifacts executing in the environment—the management of the organization is instrumented in the environment by *organizational artifacts*. The first artifact created by the `room_controller` agent (in line 10) is an `OrgBoard`. This artifact manages an organization entity and has operations to create and destroy group and social scheme entities (as used in lines 15 and 28). When a group entity is created, an instance of the `GroupBoard` artifact is created to manage it. This artifact has observable properties (such as `play/3`) and operations (such as `adoptRole/1`) related to the management of group entities. Similarly, when a social scheme entity is created, an instance of the `SchemeBoard` artifact is created.

organizational artifacts

9.4 Agents Reasoning about Their Organization

Having the organization specification and state available to the agents and giving them actions to change the organization allows us to program agents to benefit from the organization for their own goals. We illustrate these features by changing the code of `personal_assistant` agents so that they adopt a role only if: (1) they have plans for all potential goals for that role, and (2) the number of players of that role is less than the minimum.

We start the new program by importing some general inference rules (called *prolog-like rules*) available in JaCaMo. They use the organization specification as available in the *organizational belief* `specification/1`. For instance, the OS for the organized smart room example is represented by agents as

```
specification(
 os(room_org,
  group_specification( room,
   [role(controller,[],[soc],1,1,[],[]),
    role(assistant, [],[soc],1,5,[],[])],
   [],
   properties([])),

   [scheme_specification(decide_temp,
     goal(voting,performance,"",0,"infinity",[],
      plan(sequence,[
       goal(preferences,performance,"",all,"5 seconds",[],noplan),
       goal(open_voting,performance,"",all,"infinity",[],noplan),
       goal(ballot,performance,"",all,"10 seconds",
               [voting_machine_id],noplan),
       goal(close_voting,performance,"",all,"infinity",[],noplan)
      ])),
     [mission(mVote,1,2147483647,[ballot],[]),
      mission(mController,1,2147483647,
                                 [open_voting,close_voting],[]),
      mission(mAssistant,1,2147483647,[preferences],[])
     ],
     properties([]))
   ],

   [norm(n2,controller,obligation,decide_temp.mController),
    norm(n1b,assistant,obligation,decide_temp.mVote),
    norm(n1a,assistant,permission,decide_temp.mAssistant)
   ]
))
```

The inference rules allow us to make queries on this complex representation for the following predicates:

- `role_mission(R,S,M)` True when there is a norm in the OS that obliges or permits role `R` to commit to mission `M` in scheme `S`.
- `role_cardinality(R,Min,Max)` True when the cardinality of role `R` is $<$ `Min,Max` $>$.
- `mission_goal(M,G)` True when goal `G` belongs to mission `M`.

On the basis of these predicates, we add the following rules for our application:

- `role_goal(R,G)` True when role `R` is obliged/permitted to achieve goal `G`.
- `has_plans_for(G)` True when the agent has an individual plan to achieve goal `G`.
- `i_have_plans_for(R)` True when the agent has plans for all goals related to role `R`.
- `has_enough_players_for(R)` True when the number of agents playing `R` is greater than or is equal to the minimum cardinality of `R`.

Rules for these predicates are programmed as follows:

```
{ include("$moiseJar/asl/org-rules.asl") }

role_goal(R,G) :-
   role_mission(R,_,M) &
   mission_goal(M,G).

has_plans_for(G) :-
   .relevant_plans({+!G},LP) & LP \== [].

i_have_plans_for(R) :-
   not (role_goal(R,G) & not has_plans_for(G)).

```

```
13  has_enough_players_for(R) :-
14     role_cardinality(R,Min,Max) &
15     .count(play(_,R,_),NP) &
16     NP >= Min.
```

The internal action `.relevant_plans/2` used in line 8 returns in the second argument a list of plans that have as a triggering event something that matches (i.e., unifies with) the value of the first argument. The internal action `.count/2` used in line 15 returns in the second argument the number of answers for the query given as first argument.

Given these predicates, we can reimplement the role adoption of `personal_assistant` agents as follows:

```
18  +new_gr(OrgName,GroupName)
19     <-  // waits some time,
20         // so that assistants do not run this plan togheter
21         .wait( math.random(2000) );
22  
23         // focus on the OrgBoard to get its specification
24         joinWorkspace(OrgName,WspId);
25         lookupArtifact(OrgName,OrgArtId)[wid(WspId)];
26         focus(OrgArtId)[wid(WspId)];
27  
28         // focus on the GrBoard to get players
29         lookupArtifact(GroupName,GrArtId)[wid(WspId)];
30         focus(GrArtId)[wid(WspId)];
31  
32         if (i_have_plans_for(assistant)) {
33           if (not has_enough_players_for(assistant)) {
34             adoptRole(assistant)[artifact_id(GrArtId)];
35           } else {
36             .print("There are enough assistants already!");
37           }
38         } else {
39           .print("I do not have plans for role assistant!");
40           .findall(G,
41                role_goal(assistant,G) & not has_plans_for(G),
42                LG);
43           .print("No plans for ", LG);
44         }
45     .
```

Thanks to these inference rules, we have shown how we can program agents that do not blindly participate in the organization entity. An agent will participate only if able to fulfill its duties, taking into account also the current number of agents already playing the role it intends to adopt. We can note also that because the minimum cardinality of assistants is equal to 1, just one `personal_assistant` agent will be able to adopt the `assistant` role and to participate in the room temperature decision. The designer can easily change this behavior of the organization by modifying role cardinalities in the OS.

9.5 What We Have Learned

In this chapter we learned how the organization can be used to shape the set of agents into groups and roles (structure), to specify the coordinated achievement of organizational goals (function), and to assign them to agents in the system by means of norms, telling them what their obligations and permissions are. By assigning goals to roles using norms, we provide a way to abstract away from the agents and to define the functioning of the system independently from the particular agents that will be available at runtime. We have shown how easy it is to change and adapt the functioning of the MAS by changing and adapting the organization specification.

As soon as agents start to play roles, the organization entity is created, bringing obligatory and permitted goals to the agents in the context of their roles and participation in groups. The JaCaMo organization management infrastructure takes charge of the monitoring of the interpretation and execution of the organization prescriptions by the agents, taking care of the coordinated execution of the different social plans structuring the organizational goals to be achieved by the different agents participating in the organization.

Given the explicit representation of the organization specification and of the organization entity made available to the agents, the agents can undertake reasoning and make decisions on their action in the organization (e.g., adopting a role or committing to a mission) opening opportunities to handle the overall complexity of open and decentralized systems.

Exercises

Exercise 9.1 Create a new instance of the `room_controller` agent that tries to enter a room group that already has an agent playing the `controller` role. Explain the execution and then change the current implementation to allow two controllers in the organization entity.

Exercise 9.2 Change the `controller` role cardinality to $< 0, 1 >$ and describe the consequences.

Exercise 9.3 Change the `personal_assistant` agent plan for achieving goal `ballot` as follows:

a) `+!ballot  <- .wait(300).`

b) `+!ballot  <- vote([10]); !ballot.`

Will the system work properly? Why?

Exercise 9.4 Implement a plan in the `room_controller` agent that prints out all fulfilled obligations.
Hint: Consider the organizational event `oblFulfilled/1`.

Exercise 9.5 Design and implement a new policy for managing the temperature in a room: every ten minutes, the agent playing the `controller` role has to ask the preferences of the ones playing the `assistant` role and set the new temperature on the basis of that.

Exercise 9.6 Implement a plan for `personal_assistant` agents that react to the achievement of the organizational goal `close_voting` by printing the current temperature.
Hint: Consider the organizational belief `goalState/5`.

Exercise 9.7 Implement a rule `goal_role(G,R) :- ...` that can be used to infer the roles `R` of agents committed to some goal `G`.

Exercise 9.8 Implement a `personal_assistant` agent that, when asked for its preferred temperature, answers as defined in the following algorithm

```
A = set of roles of agents committed to the goal 'ballot'
B = set of all agents playing some role in A
C = set of my roles
for a in B:
   if agent a plays any role in C:
      ask agent a for its preferred temperature
      return agent a preference
return 25
```

Exercise 9.9 Revisit the distributed version of the case study in chapter 7 to make it organized using the organization defined in this chapter.

10 Integration with Other Technologies

A practical aspect that is important in tackling the engineering of real-world systems is how MAOP and the supporting technologies—JaCaMo, in this case—can be integrated with existing libraries and technologies, which are based on other paradigms. Besides implementing technical bridges, this calls also for understanding the integration from a more conceptual point of view. Integration can be in two directions: either embedding inside the MAS program some existing technology such as a library or a framework, or integrating the MAS inside some existing platform. In this chapter we discuss these aspects, first providing guidelines about integration depending on the specific technology with which to integrate and then discussing some cases in specific application domains.

10.1 Libraries, Frameworks, and Platforms

In using an MAOP approach supported by a technology like JaCaMo, there are basically two ways to integrate existing libraries and frameworks, eventually written in mainstream programming languages:

- *Agent extension* The integration is realized by extending the agents' capabilities, customizing either their architecture or their set of internal actions to wrap the technology to be integrated.
- *Artifact embedding* The integration is realized by designing and implementing new artifacts wrapping the technology to be integrated.

From a conceptual point of view, the first approach means that an agent can exploit the technology as its new personal capability. The second approach instead accounts for exploiting and interacting with the integrated technologies as resources/tools part of their environment, possibly to be shared with other agents.

Depending on the kind of technology to be integrated, it could be more effective to use one approach or the other one. The case of a library is the simplest one, because it concerns some kind of functionality provided by some module exposing some kind of interface or API, without introducing any concurrent/control issue. An example is a library for effectively managing JSON data objects. The integration of frameworks could be more complex. Generally speaking, a *software framework* is a reusable software environment meant to facilitate the development of software applications, by means of generic functionality that can be changed by additional user-written code.[1] Examples range from frameworks for developing GUI-based applications, such as JavaFX,[2] to full-fledged stacks such as Android[3]—to develop mobile apps. Different from simple libraries, software frameworks typically also introduce some kind of *control* architecture, defining how the application is executed. Sometimes software frameworks are referred also as *application platforms*, because they provide a core technology on which software developers can build programs for some specific platform.

We subsequently consider a concrete case for each category. The full source code of the examples is available on the book website.

Integrating Libraries

Libraries can be easily integrated either by means of agent internal actions or by means of artifacts; the choice may depend on what kind of functionality the library provides.

Integrating the library as an agent internal actions is straightforward when the functionality is about some stateless function extending agent capabilities. For instance, suppose that we need our agents to work with JSON data structures, exploiting existing Java-based libraries, for example, `javax.json`.[4] To that purpose, we could introduce a Jason agent library collecting a set of internal actions. Following Jason API,[5] the library can be packaged as a Java package, e.g., `json_tools`, including a class for each new actions to be added. For instance, a new internal action `json_to_list` can be provided for parsing JSON data into a Jason list:

```
package json_tools;

public class json_to_list extends DefaultInternalAction {
    public Object execute(TransitionSystem ts,
                Unifier un, Term[] args) throws Exception {
        ... // the code that implements the internal action
```

1. https://en.wikipedia.org/wiki/Software_framework
2. https://openjfx.io
3. https://developer.android.com
4. Based on the JSON Processing API JSON-P (JSR-353).
5. More about this API can be found in the Jason website http://jason.sourceforge.net and in Bordini et al. (2007).

```
    }
}
```

An example of use on the agent side:

```
test_json <-
   json_tools.json_to_list(
      "{ \"name\": \"Sofia\", \"age\": 11 }",
      L);
   .println(L). // the list L is [ name("Sofia"), age(11) ]
```

Artifacts are more convenient to use when the library to be integrated can be more useful as a tool that could be shared and concurrently used by multiple agents or (possibly stateful) tools embedding long-term heavy computations. For instance, suppose that we need our agents to have the capability to compute the Fast-Fourier-Transform,[6] to convert and analyze a signal from its original domain (time or space) to a representation in the frequency domain. We can exploit existing libraries, effectively implementing the algorithm,[7] which could be quite computationally expensive. Libraries can be based either on Java or on JVM-based languages or even other languages, exploiting the Java Foreign-Function-Interface (FFI) API to integrate them.

In this case, the library can be packaged as a `FFTCalculator` artifact, providing operations to make the transformation, making the result available as action feedback of a `transform` operation and/or as observable property of the artifact.

```
public class FFTCalculator extends Artifact {
  private FastFourierTransformer trf;

  void init(){
    trf = new FastFourierTransformer(DftNormalization.STANDARD);
  }

  @OPERATION
  void fftTransform(double[] f, OpFeedbackParam<Complex[]> r) {
    Complex[] res = trf.transform(f,TransformType.FORWARD);
    r.set(res);
  }

  // aux function to manage data structures

  @OPERATION
```

6. https://en.wikipedia.org/wiki/Fast_Fourier_transform

7. An example of the open-source library including also FFT is the Apache Commons Mathematics Library (https://commons.apache.org/proper/commons-math/).

```
  void getReal(Complex[] data, OpFeedbackParam<Double[]> r) {
   ... }

  @OPERATION
  void getComplex(Complex[] data, OpFeedbackParam<Double[]> r){
   ... }
}
```

In the following example, an agent creates a `FFTCalculator` and uses it to compute the Fast Fourier Transform given an array of `double` and prints on the console the result, as a list of complex numbers (represented by the `Complex` class).

```
1  +!test_fft
2    <- makeArtifact("calc","maop_ch10.FFTCalculator",[]);
3       cartago.new_array("double[]",[5.0, -2.0, 1.0, 2.0], Data);
4       fftTransform(Data, Res);
5       !print_result(Res). // printing: ( 6 )( 4 + 4i )( 6 )( 4-4i )
6
7  +!print_result([]).
8  +!print_result([V|T]) <-
9           cartago.invoke_obj(V,getReal,Re);
10          cartago.invoke_obj(V,getImaginary,Im);
11          !print_complex(Re,Im);
12          !print_result(T).
13
14 +!print_complex(Re,0) <- print("( ", Re, " )").
15 +!print_complex(0,Im) <- print("( ", Im, "j )").
16 +!print_complex(Re,Im) : Im < 0 <- print("( ",Re," - ",-Im,"i )").
17 +!print_complex(Re,Im) <- print("( ", Re, " + ",Im, "i )").
```

Java objects in Jason

JavaLibrary

The example is useful for talking about another issue that is quite frequent when developing and using artifacts. That is, the need to create and manipulate *Java objects in Jason*. In the `FFTCalculator`, for instance, the `fftTransform` operations want an array of `double` as first parameter and returns as output parameter a list of `Complex` objects. To this purpose, JaCaMo provides a set of internal actions packaged inside the `cartago` library,[8] referred to as *JavaLibrary* that makes it possible to instantiate Java objects (`cartago.new_obj`) including arrays (`cartago.new_array`, line 3 in the example), invoking methods on objects (`cartago.invoke_obj`). In the example, `getReal` (line 9) and `getImaginary` (line 10) are methods invoked on `Complex` objects. Detailed information about the JavaLibrary can be found in the JaCaMo documentation.

8. The internal actions to create and manipulate Java objects are part of the CArtAgO framework.

Agent Internal Actions vs. Artifacts Actions

From a technical point of view, there are important differences about how the code is executed in using either internal actions or artifacts. In using internal actions, the code is executed directly by the thread of control of the reasoning cycle, in a synchronous way. In using artifacts instead, the code, wrapped into operations, is executed by a thread of control of the environment runtime, asynchronously.

Therefore, on the one hand, performance is better in using internal actions, because there is no overhead caused by context switches, and the computational cost is equivalent to a simple (method) call. On the other hand, functionalities that could be heavy/long-term from a computational point of view should not be implemented as internal actions, because this would impact on the reasoning cycle execution and then on the reactivity of the agent to perceive events. In this latter case it is best to use artifacts, offloading onto them the computational load.

Integrating Libraries with Their Own Threads

More complex libraries could exploit asynchronous programming to provide some of their functionalities, using under the hood some threads of control to that purpose. The artifact API `beginExtSession/endExtSession` introduced in chapter 6 makes it possible for such external threads to safely change the state of an artifact, including changing observable properties and generating signals to model asynchronous events.

As an example, we consider RabbitMQ,[9] which is a well-known Message-Oriented-Middleware (MOM) or message broker. To exploit the middleware, client libraries are available in multiple programming languages. The client library basically allows sending and receiving messages, exploiting the middleware—that should be installed on every host where the MOM is used.

In this case we can define a `RMQChannel` artifact, providing operations to send messages and an observable state to perceive messages sent to the channel.

```
public class RMQChannel extends Artifact {

  private Channel channel;
  private String queueName;

  void init(String name, String host){
```

9. https://www.rabbitmq.com

```
7      ConnectionFactory factory = new ConnectionFactory();
8      factory.setHost(host);
9      Connection connection = factory.newConnection();
10     channel = connection.createChannel();
11     channel.queueDeclare(name, false, false, false, null);
12     this.queueName = name;
13
14     /* observable state */
15     defineObsProperty("lastMsg","");
16
17     /* consume msg callback */
18     channel.basicConsume(queueName, true,
19       (consumerTag, delivery) -> {
20         String message = new String(delivery.getBody(), "UTF-8");
21         beginExtSession();
22         getObsProperty("lastMsg").updateValue(msg);
23         endExtSession();
24       }, consumerTag -> {});
25   }
26
27   @OPERATION void send(String msg) {
28     channel.basicPublish("", name, null, msg.getBytes());
29   }
30 }
```

In particular, an observable property `lastMsg` is used to make observable the stream of messages sent to the channel.

An important point in the implementation, which is quite common in frameworks, is how to manage *callbacks* and *inversion of control*. In the RabbitMQ library, a callback must be registered to consume messages (lines 18–24), and it is called by the library/framework by an internal thread of the library/framework when a message is available. To update the observable property, an external session is requested to act safely on the artifact because the thread executing the code is not an environment one but a library one.

Integrating Frameworks

The management of callbacks and interaction with external threads is frequent in integrating frameworks/platforms, which may enforce the full control architecture of the application. Artifacts make it possible to have a clear separation between the specific execution logic provided by a framework and MAS execution.

As an example, we consider the integration of JavaFX, which is an open-source client application platform for desktop, mobile, and embedded systems built on

Java.[10] JavaFX provides a framework for building GUI-based applications. The framework, like almost any other GUI toolkit, is based on an event architecture and an asynchronous programming model, exploiting a single thread to execute every computation that concerns the GUI.

In this case, we can introduce an artifact `MainWindowArtifact` to represent the main application window:

```
class MainWindowArtifact extends Artifact {

    public void init() {
        defineObsProperty("button","not_pressed");
        /* setup JavaFX */
        initJFX(this);
    }

    private void initJFX(MainWindowArtifact art) {
        new Thread(() -> {
            new MainWindow().initJFX(art);
        }).start();
    }

    /* called by the JFX thread */

    void notifyButtonPressed() {
        getObsProperty("button").updateValue("pressed");
    }

    void notifyButtonReleased() {
        getObsProperty("button").updateValue("released");
    }
}
```

On the one hand, this artifact provides a high-level interface for agents to work with the main window. In this simple case, the main window has a single button that could be pressed by the human user. An observable property `button` is used to keep track and make observable to agents the state of the button (either `"pressed"` or `"not_pressed"`). On the other hand, the artifact functions as a bridge to the JavaFX subsystem, implementing (and hiding with respect to the MAS) the machinery to make the framework work. In particular, a `MainWindow` class is used to extend the JavaFX `Application` class, embedding the code to set up the GUI:

10. https://openjfx.io/

```
1  public class MainWindow extends Application {
2
3      public void start(Stage stage) throws Exception {
4          stage.setTitle("Hello World!");
5          Button btn = new Button("Say 'Hello World'");
6
7          btn.pressedProperty()
8          .addListener((observable, wasPressed, pressed) -> {
9              try {
10                 art.beginExtSession();
11                 if (pressed) {
12                     art.notifyButtonPressed();
13                 } else {
14                     art.notifyButtonReleased();
15                 }
16             } finally {
17                 art.endExtSession();
18             }
19         });
20
21         StackPane root = new StackPane();
22         root.getChildren().add(btn);
23         stage.setScene(new Scene(root, 300, 250));
24         stage.show();
25     }
26
27     public void initJFX(MainWindowArtifact art) {
28         MainWindow.art = art;
29         launch(new String[] {});
30     }
31
32     static private MainWindowArtifact art;
33 }
```

As in the previous example, a callback is managed (lines 8–19) to handle button pressed/released events, and the external interface API is used to update the observable state of the artifact accordingly. A snippet of an agent using the artifact, reacting to button press events, follows:

```
!test.

+!test
  <-  makeArtifact("mainWindow","maop_ch10.MainWindowArtifact",[],Id)
      focus(Id);
      println("ready.").

+button("pressed")
  <- println("Hello!").
```

The key point here is that agent programming is not exposed to low-level framework/platform mechanisms, and GUI events are represented at an agent level of abstraction.

Integrating MAS within Platforms

In some cases, a software framework defines its own application structure, and it is often referred in this case as a software *application platform*. In that case, it could be useful to integrate the MAS either as a component of the platform, or by executing the MAS on a different process and implementing components inside the platform that function as bridges exploiting some Inter-Process Communication (IPC) mechanism (such as sockets).

In the former case, in order to integrate directly the MAS as a component of the Java-based platform, the JaCaMo API `jacamo.infra.JaCaMoLauncher` for spawning programmatically an MAS can be used. A simple example:

```
public class LaunchMAS {
    public static void main (String args[]) throws Exception {
       jacamo.infra.JaCaMoLauncher.main(
                    new String[]{ "test-mas.jcm" });
    }
}
```

This is useful more generally to spawn an MAS from any Java (or JVM-based) program. An example of this case is discussed in the next section, about mobile apps development, where JaCaMo has been integrated with the Android framework.

In the latter case, boundary artifacts can be used to implement the bridge between the systems, embedding/hiding the use of Inter-Process Communication mechanisms and protocols to interact with the external platform.

Integration Using the Environment Interface Standard

The Environment Interface Standard (EIS) framework (Behrens et al. 2011), mentioned in section 5.4 in chapter 5, provides an effective support to connect agents and MAS written in different agent programming languages (including Jason) to environments such as games, simulators, robots, and so forth. As reported in EIS documentation (https://github.com/eishub), EIS provides the glue code to develop a connection between an agent platform and an environment. Any agent platform that supports the EIS interface can connect to any environment that implements the interface. In particular on the agent side, the EIS framework provides an API to develop the connection to *controllable entities* on the environment side; once an agent platform supports the EIS interface, it can connect to any environment that implements the interface. On the environment side, it makes it possible to develop code that allows an agent platform to connect to that environment, regardless of the specific agent platform; once the interface has been implemented for an environment, any agent platform that supports EIS can connect to it.

10.2 Mainstream Application Domains and Technologies

As mentioned previously, the need for integrating MAS with existing technologies is typical in engineering real-world applications. The integration in this case is an enabling factor to explore the application of the agent and MAS paradigm—besides the technology—to well-known application domains, exploiting the power of the agent (and MAS) level of abstraction. In this section we consider some examples in well-known application domains.

Mobile and Wearable Apps

A main application scenario for agent technologies and MAOP is given by *personal assistants*, that is, software that assists and collaborates with a human user in some task environment or context (Maes 1994). Examples in the literature of personal assistants range from scheduling joint activities (Modi et al. 2005; Wagner et al. 2004) to monitoring and reminding users of key timepoints (Chalupsky et al. 2001; Tambe 2008), searching and sharing information, and assisting in negotiation decision support (Li et al. 2006). Personal assistants can be devised for any kind of computer-based work environments, for example, office environments and desktops. Nevertheless, a main case is given by mobile and wearable computing, in which personal assistants are meant to run on mobile devices, such as smartphones, or even wearable ones, such as smart glasses.

MAOP technologies such as JaCaMo can be integrated with frameworks for developing mobile applications to develop personal assistant technologies and, more generally, agent-based applications as mobile apps. A concrete example of technologies that provide such an integration is JaCa-Android, which is an extension/specialization of JaCaMo to develop and run programs on Android devices.[11]

Generally speaking, JaCa-Android is not just a simple porting but provides a conceptual and practical blueprint to design and develop a mobile app as an agent-based system. From an architectural point of view, at a coarse-grained level, an Android application is made of *activities*—the app components serving as entry point for interacting with the user, representing a single screen with a user interface—

11. https://developer.android.com

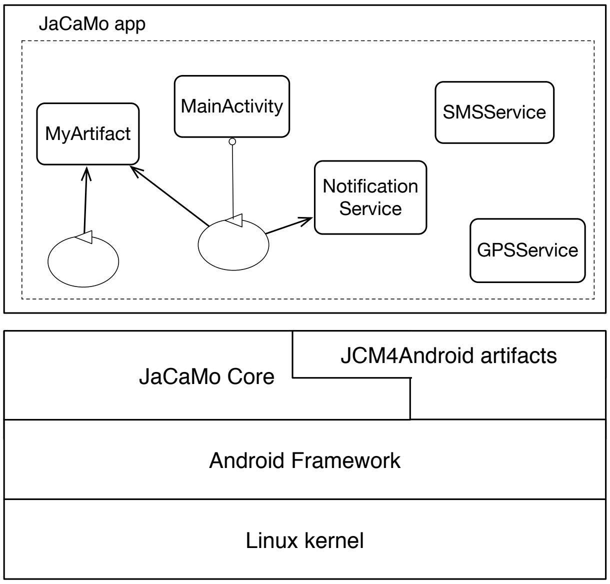


Figure 10.1
Architectural levels of an Android app based on JaCaMo.

and *services*—as general-purpose entry point for keeping an app running in the background for all kinds of reasons. Whereas OOP is used as the underlying programming paradigm (using languages such as Java, Kotlin, or C++), the control architecture is event driven. There is only one active activity at a time—the foreground activity—which is executed by a single thread implementing an event loop. The application logic is fragmented in callbacks, asynchronous tasks, and Java-based threads to be used for long-term jobs.

By using JaCa-Android, an Android app is designed as an MAS—like any JaCaMo program—in which agents are used to encapsulate the control logic of the applications. Figure 10.1 shows the architectural levels of a JaCaMo mobile app. A specific library of artifacts is provided, wrapping the basic functionalities and services provided by the mobile/wearable computing environment. Among the others:

- Artifacts to implement activities, in particular to model only the User Interface (UI)—not the control part, which is stored in agents observing/using such activities.

- Artifacts representing device sensors and actuators, and services to acquire information about the user context (such as the location). Examples include `BatteryService`, `GPSService`, and `SMSService`.
- Artifacts to trigger and interact with other activities and apps available on the device.

Figure 10.2 shows a simple example of an agent that shows a notification when an SMS sent by a specific sender is received. The agent exploits a couple of artifacts provided by the JaCa-Android platform, namely, the `SMSService`, to manage SMSs, and a `NotificationService`, to show notifications on the status bar. The artifact `SMSService` generates an `sms_received` each time a new SMS arrives. When a new SMS arrives from the sender that the agent has to track, the agent displays the content either on the main activity of the app if the application is in the foreground, or on the status bar, if the app is not in foreground. The main activity is represented by the `MainActivity` artifact (shown in figure 10.3), extending the base class `ActivityArtifact`, which is part of the JaCa-Android API. In this artifact, the `viewer_state` observable property is set to either `displayed` or `not_displayed` depending on the state of the main activity (i.e., of the app), if it is either in the foreground or not. Instead, `showNotification` is an operation provided by the `NotificationService`.

Compared with the Android framework, JaCa-Android provides an upper level of abstraction meant to simplify the development of personal assistants, as Jason agents that exploit artifacts to observe and interact with users and their context. A real-world example in the health care context is discussed by Croatti et al. (2018), where a personal agent is used to assist medics of a trauma team in doing precision tracking during a trauma resuscitation.

Web Technologies

The integration of agent and web technologies is useful both to exploit agents and MAS in the design and development of smart service-oriented architecture-based systems (Huhns and Singh 2005), as well as to exploit web standard protocols to enhance interoperability of agent-based applications.

In fact, the integration can be tackled at different levels of the web technology stack, including the semantic level. In the following, we consider only the enabling level, the bottom one. At that level, basically there are two main cases:

- client side — enabling agents to exploit existing web-based services and web applications;
- server/service side — exploiting agents to implement web-based services and web applications.

```
!warn_about_sms_from("999999").

+!warn_about_sms_from(Src) <-
  +source_to_track(Src);
  !setup.

+sms_received(Src, Msg) : source_to_track(Src)
  <-  !notify_sms(Src, Msg).

+!notify_sms(Src, Msg) : viewer_state("displayed")
  <-  displayNewSMS(Src, Msg).

+!notify_sms(Src, Msg) : viewer_state("not_displayed")
  <-  showNotification(Src, Msg).

+!setup <-
  makeArtifact("sms-receiver", "jaca.android.SMSService", [], SMSid);
  focus(SMSid);
  makeArtifact("notificator", "jaca.android.NotificationService", []);
  lookupArtifact("mainActivity", IdViewer);
  focus(IdViewer).
```

Figure 10.2
An agent tracking SMS messages on JaCa-Android.

```
public class MainActivity extends ActivityArtifact {
  void init() {
    defineObsProperty("viewer_state", "not_displayed");
  }
  @OPERATION
  void displayNewSMS(String source, String msg){
    Viewer mViewer = (Viewer) getActivity("viewer");
    mViewer.append(source, msg);
  }
  @INTERNAL_OPERATION
  void onStart(){
    ObservableProperty o = getObsProperty("viewer_state");
    o.updateValue("displayed");
  }
  @INTERNAL_OPERATION
  void onStop(){
    ObservableProperty o = getObsProperty("viewer_state");
    o.updateValue("not_displayed");
  }
}
```

Figure 10.3
Artifact used by the agent to interact with the user.

Client side At the basic level, this case accounts to equip agents with actions that make it possible to execute requests using the web standard technology. In an MAOP-based approach, a straightforward way to do it is to design a proper boundary artifact that implements those actions as operations. Artifacts can play a twofold function:

- encapsulate and hide technical details that concern web technologies, including aspects that concern the invocation semantics; and
- bridge the representation of the data exchanged, so as to make it more appropriate on the agent side.

A simple example of an artifact enabling the interaction with a web resource/service using a REST style follows:

```
public class WebResource extends Artifact {

  private CloseableHttpClient client;

  void init() throws Exception {
    // creating a factory and connection manager for HTTP(S)
    SSLConnectionSocketFactory sslsf = ...
    BasicHttpClientConnectionManager connectionManager = ...
    // creating the client object
    client = HttpClients.custom()
      .setSSLSocketFactory(sslsf)
      .setConnectionManager(connectionManager).build();
  }

  @OPERATION
  void get(String uri, OpFeedbackParam<String> res){
    HttpGet req = new HttpGet(uri);
    CloseableHttpResponse response = null;
    try {
      response = client.execute(req);
      if (response.getStatusLine().getStatusCode() >= 300) {
        failed("error");
      } else {
        res.set(extractResponse(response.getEntity()));
      }
    } catch (Exception ex) { ...  }
    } finally { ... }
  }

  @OPERATION
  void post(String uri, String payload,
                OpFeedbackParam<String> res) { ... }
  private String extractResponse(HttpEntity entity) { ... }
}
```

The artifact implementation is based on the Apache HTTPComponents library.[12] In this simple case, the data format used to specify the payload for the request and the response is simply a string.

A snippet of an agent plan using the artifact follows:

```
+!test_web
  <- makeArtifact("web","maop_ch10.WebResource",[],Id);
     print("posting a new dweet on maop-book thing...");
     Msg =  "{ \"msg\": \"hello, world!\" }";
     post("https://dweet.io:443/dweet/for/maop-book",Msg, _)
     println("done.");
     println("getting the latest dweet on maop-book...");
     get("https://dweet.io:443/get/latest/dweet/for/maop-book",Res)
     println(Res).
```

The agent uses the artifact to post a message on the dweet web messaging service[13] on a thing called `maop-book` and then to get the last message posted.

Different kinds of artifacts for the web can be designed according to the need. For instance, instead of using a REST-based style, an artifact can implement a SOAP-based style, providing facilities to manage the full WS-* stack. The examples shown so far provide a bridge at the web technology level of abstraction. This makes it possible to use these artifacts to interact with any kind of web-based services.

A further general approach consists of using artifacts to directly model the (web-based) service, providing an interface with operations and observable properties that are conceived at the *domain level*. A simple example follows, a `MapArtifact` providing services about geographical maps, exploiting under the hood the Google Map API:[14]

```
public class MapArtifact extends Artifact  {
  private GeoApiContext context;

  void init(String apiKey) {
    context = new GeoApiContext.Builder().apiKey(apiKey).build();
  }

  @OPERATION
  void getGeoCoordinates(String place,
```

12. https://hc.apache.org
13. http://dweet.io
14. https://developers.google.com/maps/documentation

```
              OpFeedbackParam<Double> latit,
              OpFeedbackParam<Double> longit) {
    try {
      GeocodingResult[] results =
            GeocodingApi.geocode(context,place).await();
      latit.set(results[0].geometry.location.lat);
      longit.set(results[0].geometry.location.lng);
    } catch (Exception ex) {}
  }
  ...
}
```

An example of a plan using the artifact to request the latitude and longitude of some place (an example of a goal could be `!test_map("Paris, France")`):

```
apiKey("AIzaSyDiPt735ULDnFl9Iwz4ZyeEzt1LKlxOVyE").
...
+!test_map(Place) : apiKey(APIKey)
  <- makeArtifact("map","maop_ch10.MapArtifact",[APIKey]);
     println("Requesting information about: ",Place,"...");
     getGeoCoordinates(Place,Lat,Long);
     println("Results - latitude: ",Lat,", longitude: ",Long).
```

To make the example work, a valid API key must be specified.[15] In this case, artifacts make it possible to model the interaction with existing (web) resources and services at an upper level of abstraction, focusing on the facilities provided by the service. This makes it possible in principle to reuse the same artifact—or, an artifact with the same interface—regardless of the specific web service API used inside the artifact.

As a final remark, it is worth noting that from a technical point of view, it would be possible to use internal actions to interact with web resources/services. However, from a conceptual point of view, internal actions are meant to affect or access the *internal* state of the agent, not the environment. This is the main reason why we used artifacts rather than internal actions. Nevertheless, in the case that a web resource/service would be conceptually part of the agent—for example, implementing a kind of auxiliary memory—then internal actions would be a good choice as well.

Service side In this case, boundary artifacts can be used to mediate the interaction between users making web-based requests and agents serving the requests. In particular, an artifact can be useful to wrap the service side machinery to accept the web requests and make them available to agents, enabling them to process the requests and eventually send responses. An example follows:

15. The API key can be obtained by registering on the Google Cloud platform.

```
public class RESTWebService extends Artifact {

    private Vertx vertx;
    private Router router;
    private HttpServer server;

    void init() throws Exception {
        vertx = Vertx.vertx();
        router = Router.router(vertx);
        router.route().handler(CorsHandler.create("*")
            .allowedMethod(io.vertx.core.http.HttpMethod.GET)
            .allowedMethod(io.vertx.core.http.HttpMethod.POST)
        ...
        router.route().handler(BodyHandler.create());
    }

    @OPERATION
    void start(int port) {
        server = vertx.createHttpServer()
            .requestHandler(router)
            .listen(port, result -> {
                if (result.succeeded()) {
                    log("Ready.");
                } else {
                    log("Failed: "+result.cause());
                }
            });
    }

    @OPERATION
    void acceptGET(String path) {
        router.get(path).handler((res) -> {
            this.beginExtSession();
            this.signal("new_req", "get", path, res);
            this.endExtSession();
        });
    }

    @OPERATION
    void acceptPOST(String path) { ... }

    @OPERATION
    void sendResponse(RoutingContext ctx, String res) {
        ctx.response().end(res);
    }
    ...
}
```

The `RESTWebService` artifact exploits the `vertx` technology[16] to set up an event-driven web service based using a REST style. The artifact provides `acceptXXX` operations to configure the requests to serve, which are made observable to agents using signals. In addition, it provides an operation to send responses and close the request (`sendResponse`). On the agent side, in the following example an agent uses the artifact to set up a REST web service (on port 8090), accepting GET and POST requests on a `count` resource. Each time a new request arrives, the agent reacts and serves them, keeping track of the value of the count using a belief:

```
+!test_web_service
  <- makeArtifact("web service","tools.WebService",[],Id);
     acceptGET("/api/count");
     acceptPOST("/api/count/inc");
     focus(Id);
     +count(0);
     start(8090).

+new_req("get","/api/count", Req) : count(C)
  <- .concat("{ \"count\": ", C, " }", Reply);
     println("GET req on /api/count. Replying: ",Reply);
     sendResponse(Req, Reply).

+new_req("post","/api/count/inc", Req) : count(C)
  <- getBodyAsJson(Req,Body);
     C1 = C + 1;
     -+count(C1);
     .concat("{ \"count\": ", C1, " }", Reply);
     println("POST req on /api/count/inc. Replying: ",Reply);
     sendResponse(Req, Reply).
```

As in the client side case, the second level is about using the artifact to model directly the service at the domain level, however functioning as interface (or adapter), so as to allow for using agents to encapsulate the service application logic.

MAOP and the Web of Things

So far, we have discussed two sides of the integration of agent and web technologies: on the client side, agents consume services; on the server side, agents are used to implement services. A third viewpoint on this problem is a conceptual integration between MAS and the web (Ciortea et al. 2019): instead of examining agents and services side by side and using web tech-

16. http://vertx.io/

nologies to bridge different implementations, in this perspective MAS use the web architecture as the underlying glue that interconnects all entities in the MAS (agents, artifacts, organizations, and so forth) and allows them to interact with one another.

Central to this viewpoint is to consider the environment as a first-class abstraction in the MAS (Ciortea et al. 2019). In the conventional view, in which MAS are composed only of agents, conceptually there is little room left to use the web as anything more than a transport layer for agent message. However, if we consider also the environment as a first-class abstraction in the MAS, then the web can provide an application layer to support all sorts of environment-mediated interactions, such as interactions between agents and devices using the W3C Web of Things Thing Description (https://www.w3.org/TR/wot-thing-description).

Robotic Integration

In this section we discuss a particular solution to embed JaCaMo inside robots, in particular robots that support ROS (Robotic Operating System).[17] ROS simplifies the access to the hardware using abstract *topics* that our agents can listen to (to obtain information from the hardware) and publish (to control the robot). Of course, we could have an artifact that translates some topics to observable properties and other topics to operations. Another option is to customize the agent architecture, changing how agents perceive and act on the environment. Whereas the former solution does not change the usual way agents perceive and act, the latter allows more control and optimization in the integration. The project Jason-ROS[18] explores this latter approach; it is presented below.

The example used in this section considers a simple Turtle Bot that provides the following topics (figure 10.4):

pose Used to get the current position of the robot (perception);
cmd_vel Used to set the velocity and direction of the robot (action); and
set_pen Used to set the color the turtle is painting (action).

The customized agent architecture implemented in Java overrides two methods: `act` (called for every action the agent decides to execute) and `perceive` (called at

17. https://www.ros.org
18. https://github.com/jason-lang/jason-ros

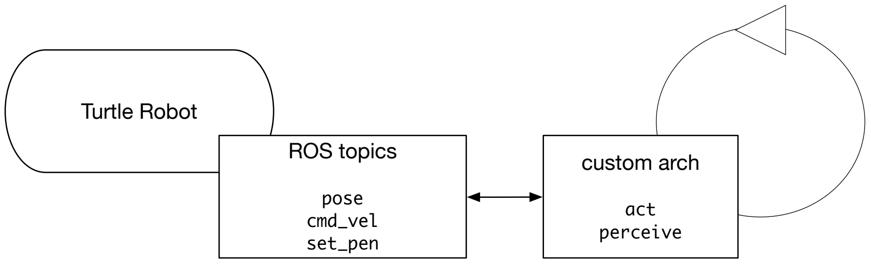


Figure 10.4
Customized agent architecture to integrate ROS.

the beginning of every reasoning cycle).[19] The Jason-ROS implementation of these methods uses configuration files to state how topics are mapped into beliefs and actions. The perception is configured as follows:

```
[pose]
name = /turtle1/pose
msg_type = Pose
dependencies = turtlesim.msg
args = x,y,theta
```

It maps the ROS topic `/turtle1/pose` (of type `Pose`) into beliefs like `pose(x,y,theta)`. Other parameters can be specified to set the frequency used to update this belief. Actions are configured as follows:

```
[cmd_vel]
method = topic
name = /turtle1/cmd_vel
msg_type = Twist
dependencies = geometry_msgs.msg
params_name = linear.x, linear.y, linear.z, \
              angular.x, angular.y, angular.z
params_type = float, float, float, float, float, float

[set_pen]
method = service
name = /turtle1/set_pen
msg_type = SetPen
```

19. More about customizing an agent architecture is found in Bordini et al. (2007) and in the Jason website http://jason.sourceforge.net.

```
dependencies = turtlesim.srv
params_name = r, g, b
params_type = int, int, int
```

This configuration maps the execution of external actions like `cmd_vel(1.0, 5.0, 2.0, 3.0, 2.0, 1.5)` and `set_pen` into a publishing on ROS topics as `/turtle1/cmd_vel` (of type `Twist`) and `/turtle1/set_pen` (of type `SetPen`). Actions different from `cmd_vel` and `set_pen` are not handled by the architecture and are processed as usual in JaCaMo. All actions handled by custom architectures are considered by Jason as external actions, and thus their execution is asynchronous, as are operations on artifacts.

To use this particular agent architecture, the agent declaration in the application should include an `ag-arch` entry:

```
mas turtle {
  agent t {
    ag-arch: jasonros.RosArch
  }
}
```

From the agent programming perspective, there is no difference. We continue to use beliefs and actions as usual. For example:

```
+pose(X,Y,T) <- .print("I am at ",X,",",Y).
+!paint(red) <- set_pen(255,0,0).
```

Custom agent architectures is a solution for integration when a project requires fine-grained control on perception and action, for instance, when the concurrency model from CArtAgO is not required (or desired) while keeping the asynchronous execution of external actions. More control might imply more effort, too. For example, the implementation of a new `perceive` method may require an implementation of the belief update function (BUF), which may not be trivial.

10.3 Integration with Other Multi-Agent System Platforms

When we think about multi-agent systems as *open systems*, an interesting issue is how different agents and MAS, developed using heterogeneous programming languages, technologies, platforms, can work together, in the same *systems of systems* (Nielsen et al. 2015), which may also include nonagent technologies. Such a scenario calls for different levels of *interoperability*. In an MAOP perspective, we can identify two main kinds of approaches.

The first approach is exploiting a common agent communication language (ACL), introduced in chapter 7, enabling agents written using different technologies and running on different platforms to talk together. As previously mentioned in this book, the most popular ACLs are FIPA ACL (by the Foundation for Intelligent Physical Agents, a standardization consortium) and Knowledge Query and Manipulation Language (KQML). Both define a set of performative verbs, also called communicative acts, and their meanings (e.g., ask-one). JaCaMo supports both ACLs; in particular FIPA ACL is supported by means of the integration with the JADE platform. Sharing a common ACL is just an enabling factor for interoperability; to make agents understand each other, they need also to have a common *ontology*, besides speaking the same language. Ontologies allow for defining a common syntax and semantics about the content of the messages exchanged by the agent.

The second approach is based on sharing a common environment that could be joined by agents written using heterogeneous languages and technologies. The CArtAgO framework on which JaCaMo is based was conceived to be exploited with different agent models and technologies, providing bridges to specific agent programming languages and technologies (Ricci et al. 2008). In this case, the heterogeneous agents interact and interoperate indirectly by exploiting the shared artifact-based environment. As in the case of ACL, this is just an enabling factor. In order to achieve full interoperability, ontologies would play an essential role, in this case, to have a common understanding about artifact interface and functionalities. In the A&A model, the information about what are the functionalities of an artifact and how to use them is meant to be provided by the *artifact manual*.

A further approach that enables heterogeneous agents to work and interoperate in the same environment is provided by EIS framework, mentioned in a previous section. In this case, EIS provides a common model not of the environment but of the *interface* to access and work in some environment, without constraining the specific environment (meta)model to be adopted.

Ontologies and Agents

Different models and languages have been proposed in literature for the interchange of knowledge among different agents and to define ontologies.

An example is the Knowledge Interchange Format (KIF), a language proposed by Stanford AI Lab, designed for enabling the interchange of knowledge among disparate computer systems, created by different programmers, at different times, in different languages, and so forth. In the context of agents, KIF has been used primarily as a language to describe the message content in ACL, KQML in particular (Finin et al. 1994).

The definition of common ontologies is a main aspect of the semantic web (Berners-Lee et al. 2001) which is an extension of the World Wide Web through standards that promote common data formats and exchange protocols on the web. These includes the Resource Description Framework (RDF) and the Ontology Language (OWL), the latter specifically conceived to represent rich and complex knowledge about things, groups of things, and relations between things. The semantic web effort is essential for bringing communicating multi-agent systems to the World Wide Web, integrating intelligent agent technology and ontologies (Hendler 2001).

In the context of agent programming, ontologies can be used both to define the content of message exchanged and interaction protocols and, more generally, to represent knowledge inside the agents, that is, agent beliefs in the BDI model. The use of ontologies in defining interaction protocols is among the main features of the JADE platform (Bellifemine et al. 2007). JASDL (Klapiscak and Bordini 2009) is an example of proposal in literature in which the Jason agent programming language is extended so as to use OWL to represent beliefs, enabling features such as plan trigger generalization based on ontological knowledge and the use of such knowledge in querying the belief base. More about agent programming and ontologies can be found in chapter 11, in the Semantics and Reasoning section.

11 Wrap-Up and Perspectives

In this final chapter, we provide an overview of some main research perspectives that concern MAOP. After recalling the key points that characterize the MAOP approach, we focus on AI and discuss how AI-related classic problems can be posed in a different way when considering an MAOP perspective. Then, we move on to software engineering, discussing how techniques for engineering complex software systems can be supported by MAOP and, more generally, an agent-oriented approach to software engineering. Throughout the chapter, we provide references to research work that has already been done and published, and mention future directions of research.

11.1 The MAOP Viewpoint—Wrap-Up

In this book, we have presented the MAOP approach with a focus on each of the dimensions that take part in its definition (agent, environment, and organization dimensions in chapters 4, 5, and 8), making clear what programming abstractions they offer. We have also shown how these abstractions integrate all together for programming modern software applications in which an increasing level in the autonomy of the interconnected software systems is required (see figure 11.1).

A rich and flexible flow of information and control among the software entities instantiating each of the independent dimensions is possible, thanks to the dynamic relations that loosely connect them.

- The *communicate* relation connects the agent dimension to itself. This relation represents the ability for agents with the same or different architectures to communicate with each other. Such direct interaction among agents is realized thanks to the performative verbs (see the technology corner on page 93) and the common language used to express the contents of messages in terms of sharing beliefs, goals, events, and plans (see the research corner on page 186).
- The *perceive* and *act* dynamic relations connect the agent dimension to the environment one. They represent perception and action of the agents on a shared

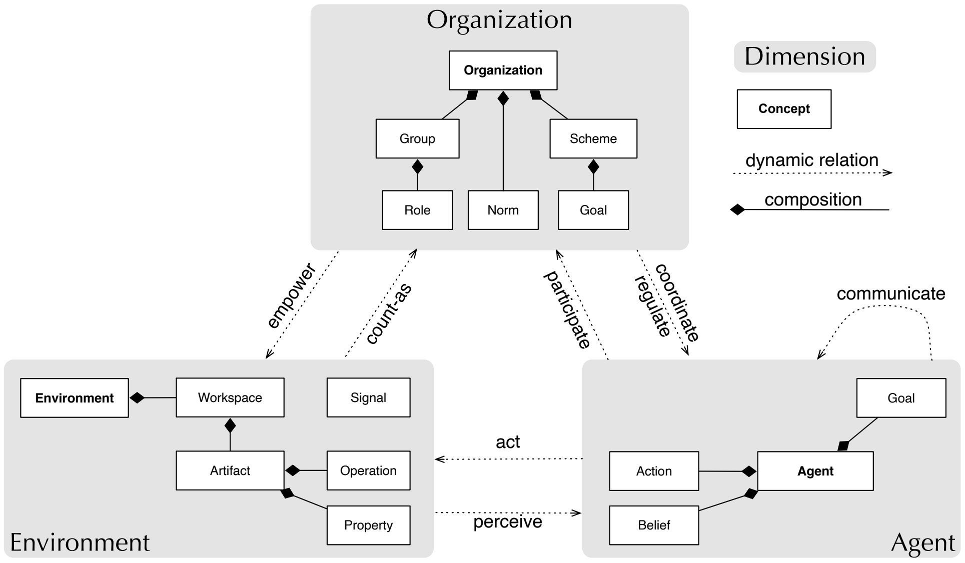


Figure 11.1
MAOP dimensions.

environment, and thus may be involved in indirect interaction, which is the other facet of interaction among agents.

Dynamic relations between agents and the environment have been investigated from a practical point of view in chapters 6 and 7.

- The dynamic relations *participate*, *coordinate*, and *regulate* connect the organization and agent dimensions. Presented in chapter 9, they represent the means for agents to actively participate in the coordination and regulation patterns imposed on them by the organization to which they belong.
- The dynamic relations *empower*, *count-as* that connect the environment and organization dimensions have been presented in the research corner on page 140 and are ongoing work.

With the case study explored in chapters 6, 7, and 9, we have first shown how to program agents and the shared environment where they are situated. We have then discussed the programming of coordinated behavior among those autonomous entities. To that purpose, we have presented how to use the programming abstractions of each of the dimensions that take part in the definition of the MAOP approach. As done by Boissier et al. (2019), we have shown how the same coordination pattern to achieve the choice of the preferred temperature can be programmed by exploiting direct communication, shared environments, or agent organizations.

All along these chapters, we discussed how alternative solutions result from different synergies between the available dimensions, that is, an emphasis on a particular dimension generates a particular solution for the given problem. We discussed the benefits and limitations of each of those solutions to approach the same problem. The JaCaMo platform has been used as a development tool for that exercise. This way, we illustrated MAOP from a practical point of view and discussed the development of a simple system integrating the agent, environment, and organization dimensions.

11.2 MAOP and Artificial Intelligence

Each of the following sections is dedicated to the description of some classic AI-related problems and how MAOP helps to revisit them, or to the description of a technology and how it in turn is helping further progress MAOP to support the development of ever-increasing system complexity. We chose to focus particularly on the following problems: semantics and reasoning, planning and acting, adapting and learning, and argumentation.

Semantics and Reasoning

Semantics is all about the meaning of symbols. In the context of an MAOP approach, the range of symbols and expressions may be associated with a large collection of entities in the real world. Such entities may refer to the autonomous agents or to the nonautonomous entities in the environment such as tools or resources. However, the symbols may also relate to a *service* level, a more abstract level used to describe the agents, tools, or resources in terms of service profiles, categories of objects with which it is possible to interact in terms of agent-to-agent interaction or in terms of agent-to-tools interaction. Finally, they may also pertain to a *policy* level, an even more abstract level on which the symbols are no longer used to describe the content of interaction but to describe how to coordinate and regulate the interaction among the entities (e.g., groups, social schemes, or norms defining the organization). For each of these levels, agents are required to be capable to compute meaningful correspondences between the symbols that they use, so as to interact and coordinate effectively while acting in the environment.

As noticed by Argente et al. (2013), while aligning their representations about the entities, services, or policies, agents may encounter different types of conflicting issues:

1. Lack of same vocabularies or conceptualizations although using a common formal language (e.g., for specifying their ontologies or for sharing information in the content communication language);
2. Different languages and, possibly, different vocabularies and conceptualizations; and
3. Semantics that is implicit and informal, being hardwired into the agents' decision-making mechanisms, the reasoning cycle, or the agent program.

It is clear that moving from the first to the third type corresponds to an increase in the difficulty of achieving semantic agreement and understanding among the agents.

From the environment point of view, the use of semantic technologies is related to the descriptions of artifacts, which in existing work have been called artifact manuals (Acay et al. 2009). Furthermore, as far as the environment acts as an interaction mediator, the environment may provide the appropriate facilities (e.g., services, repositories, and information sources) for the agents to access semantics descriptions (e.g., ontology services [Mascardi et al. 2014; Freitas et al. 2015]) or to reach semantic agreement. Nevertheless, as the difficulty increases (something expected in open settings), the role of the environment toward establishing semantic correspondences should become more important (Argente et al. 2013). The environment provides a common ground where all the agents are deployed. The environment becomes the actual and common context to all the agents. This leads to *situated cognition* in which knowledge exists inseparably from context. Therefore, aligning semantics and establishing agreements can be codetermined by the agents and their context instead of an objective alignment or combination of knowledge. This opens for increasingly effective performance.

At the agent level, besides using semantic techniques to describe the agents in a system (as we discuss subsequently), semantic techniques can also be used for reasoning within agents. Building on various previous lines of research on combining AgentSpeak with ontological reasoning, Mascardi et al. (2014) proposed CooL-AgentSpeak which, on top of the various advantages for ontological reasoning with regard to plan selection, querying the belief base, and various other aspects, it also added support for the use of ontology alignment services.

Similarly, at the organizational level, we can use semantic technologies for the description of organizations (e.g., as in Semantic Moise [Zarafin et al. 2012]), but conversely, organizations can be used as means for structuring semantics (e.g., attaching a set of ontologies to organizations, meaning that in those organizations using a particular ontology is mandatory, using another ontology is forbidden, and so forth).

Finally, Freitas et al. (2015) give semantic descriptions for all JaCaMo dimensions, putting forward an approach to facilitate the modeling of JaCaMo systems. By extending a top ontology to the specific system to be developed, the user can take advantage of the specified concepts when coding the system, through the use of techniques such as drag-and-drop and automatic generation of some code structures. The approach also allows the use of ontological reasoning to check for some aspects of code consistency.

Planning and Acting

As summarized by Trentin et al. (2019), the cognitive functions necessary for agents to be able to act upon and perceive a dynamic world are commonly grouped in what are called *planning* and *acting* functions (Ghallab et al. 2016). The latter is a deliberation function that has direct contact with the external world, responsible for following plans (i.e., a set of ordered actions built to achieve goals), and reacting to context (with every input an agent checks whether plans are still feasible and chooses appropriate available actions to be executed), all driven by previously chosen goals. The planning function is another deliberation function that complements the acting one by providing it with a set of plans customized to achieve previously defined user goals. The inputs to this planning function are a set of goals and an operational model of the actions available to the agent. Such actions will be combined by the planner to create plans capable of achieving goals given a context. There can be also a descriptive model, which is a set of abstract actions used by the planning function to create plans to achieve goals, whereas the operational model is a set of low-level commands used by the acting function to execute the abstract actions present in plans.

Creating new plans at runtime can clearly make a significant impact on an agent's ability to autonomously operate in unpredictable environments. Reconsidering this global problem in the context of an MAOP approach takes into account the following problems and opportunities.

Defining the descriptive model of actions to be used by the planning function requires having a description of the possible actions that are made available to the agents. In MAOP, this corresponds to the set of operations in artifacts that could be deployed and used by the agents in the workspaces that they have joined. Two main research questions then arise:

1. The existence of a "manual" with a description of an artifact usage interface allows agents to reason about it and build plans to use those descriptions of operations in the achievement of their goals (Acay et al. 2009).
2. One can consider introducing in the descriptive model a set of meta-actions related to the creation of artifacts, the joining of workspaces, the focusing on

artifacts—that is, all sorts of actions that deal with the management of artifacts themselves. Thus, one can enrich planning with actions related to the deployment and configuration of the working environment for the agents.

The majority of existing approaches, however, provide planning capabilities that consider exclusively at the agent dimension. Such approaches use the great variety of planning techniques that exist in the literature. These are, for instance, first-principles planning (FPP), which is, informally, the creation of new plans, based upon an action theory, that is, a predefined set of actions, to achieve goals (Xu et al. 2018; Silva et al. 2009); Markov Decision Processes (MDP) (Bellman 1957); and Hierarchical Task Network (HTN)—a well-known planning technique in which compound tasks are decomposed into simpler ones until tangible actions that can be directly executed are obtained (Herzig et al. 2016). The question to investigate is how to integrate in these agents the ability to plan, as well as the ability to reason about the other agents to prevent conflict at the planning phase (epistemic reasoning and planning).

Pioneering work on integrating planning into BDI agent programming appeared in work by Sardiña et al. (2006, 2011), which introduced a language named CANPlan. The Refinement Engine for Acting and Planning (REAP) has a similar direction (Ghallab et al. 2016). These approaches aim to incorporate HTN-style planning into BDI-like agents, ultimately allowing agents to perform online planning and acting. Another approach is HTN Acting (Silva 2018a,b), whose approach also combines HTN planning with BDI behavior, that is, performing interleaved deliberation, acting, and failure recovery. By adapting HTN planning semantics, HTN acting does the opposite of REAP and CANPlan, that is, adapting BDI agents.

An example is the embedding of FPP in BDI agents (Xu et al. 2018; Silva et al. 2009), giving it some planning capabilities. However, combining BDI agents and HTN planning appears more promising, perhaps because of the similarity between BDI agent plans and HTN methods, which implies that an HTN planner can reuse the domain knowledge available in BDI agents' plan libraries, and is hence quite apropriate for online planning, as done by Cardoso and Bordini (2019); this work is subsequently discussed further.

An agent that can create new plans is capable of adapting itself in an online, goal-driven fashion. But its behavior might be completely ignorant of the presence of other intelligent agents, ultimately simply sensing other agents' actions in the world and considering them simple changes to be adapted to. Therefore, we are interested in allowing interaction and coordination among decentralized agents, and another major challenge is planning for human-agent interaction.

At the organizational level, social schemes represent plans at the collective level. The architecture presented by Ciortea et al. (2018) shows a possible approach using the organization dimension as a means of coordinating agents: the organization has social schemes (sets of collective goals) which are used to allocate available agents that are capable of executing such goals. The allocation is done by giving roles to agents. Another approach, the Decentralized Online Multi-Agent Planning (DOMAP) presented by Cardoso and Bordini (2019, 2017), uses the environment dimension to coordinate agents. That approach breaks down the process of multi-agent planning into three parts: goal allocation, individual (HTN) planning, and execution using coordination artifacts. Hence there are three special artifacts: a *task board*, a *contract net board* or something similar to do task allocation, and an artifact with *social laws* to do runtime coordination. The bidding for goal allocation uses heuristics based on metrics taken from the agent's plan library. The social laws are special artifacts that impose rules in order to resolve conflicts at runtime. Although that work can be used, for example, when a JaCaMo organizational scheme fails to achieve the main goal, much still remains to be done in terms of planning for JaCaMo systems that considers all three dimensions.

Learning and Adapting

We have seen that the agent architecture used in JaCaMo and more generally the multiple dimensions of abstractions available in it provide many opportunities for taking advantages of off-the-shelf AI techniques. Besides ontological reasoning and automated planning, there are many opportunities for the use of (currently so widespread) machine learning techniques. The most directly usable way, in particular for work on machine learning for computer vision, is to simply use them to provide percepts for agents. In order to do so we need only the result of image processing to be represented in the particular symbolic way that our agents use to represent percepts.

Some of the earliest ideas on learning and adaptation for BDI agents appeared in work by Guerra-Hernández et al. (2004) and by Airiau et al. (2009). Singh et al. (2011, 2010a,b) model plan contexts in BDI agent plan libraries similar to those used in this book as decision trees, selecting a plan among the various available plans to achieve a goal is done probabilistically, and machine learning techniques are used to improve such selection; the idea is to allow agents to learn from experience which among the various available plans is more likely to succeed at given circumstances. More recent and even more directly related to our MAOP approach, Ramirez and Fasli (2017) put forward an approach to *intentional learning* focusing on plan acquisition (i.e., creating new know-how to adapt the agent behavior at runtime) implemented in Jason.

In the literature, learning techniques have been used mainly to improve agent adaptation, by means of, for example, plan or action selection *at runtime*. A further perspective has been recently introduced by Bosello and Ricci (2019), in which the value of learning is explored also *at design/development time*, as a method to automatically create plans instead of writing them by hand. The basic idea is that to design an agent capable to achieve some goal g, an agent developer (using Jason, for instance) may decide to find it more convenient not to write plans by hand but to allow the agent itself to learn the best way to achieve the goal. This could be done by means of, for example, a reinforcement learning (RL) process (Sutton and Barto 2018), based on some simulation environment designed for that purpose. In that perspective, developing an agent would mean integrating both hand-coded plans and plans learned by the agent itself through experience, in a training stage during agent development. The same idea could be applied as well at the interacting agents and organization level.

Argumentation

Argumentation theory (Walton et al. 2008; Simari and Rahwan 2009) has become an important research theme in artificial intelligence as it provides principled means for both reasoning about conflicting information (which is pervasive in multi-agent systems and more generally in the real world) as well as for autonomous agents to reach agreements through communication. In regard to an individual agent reasoning about conflicting information available to it based on argumentation techniques, it is useful to add explicit defeasible reasoning rules as part of the agent belief base, as done by Panisson and Bordini (2016). This allows conclusions and justifications used in arguments that an agent put forward to be an integral part of the agent's usual reasoning mechanisms. That is, those arguments are handled by the existing belief base structure of the agent. An alternative approach that is less integrated with the existing belief base but also implemented for Jason and hence applicable to JaCaMo was proposed by Zavoral et al. (2014). The implementation by Panisson and Bordini (2016) is based on defeasible Prolog (Nute 1993, 2001), so the approach for reasoning about conflicting information taken by our agents is equivalent to what in argumentation theory is known as *grounded semantics*, as shown by Governatori et al. (2004).

On the agent communication side, we have given formal semantics to speech acts that are specific to argumentation (Panisson et al. 2014), and therefore more expressive than the usual agent communication processes; the semantics was implemented using the straightforward means for the definition of new performative verbs supported by JaCaMo. That approach allows agents to make claims, justify how they are able to conclude particular claims, question claims or justifications made by other agents, and so forth. Making such dialogue moves generates commitments for the participating agents, in the sense that an agent becomes committed to defending the claims it has made throughout the interaction process (i.e., justifying them by communicating the reasoning that allowed that agent to reach those conclusions), unless of course on the basis of new information exchanged with other agents the agent chooses to retract those previous claims. In our JaCaMo framework, artifacts are used to keep track of the *commitment store*. Normally, we want to have mechanisms to ensure such multi-agent communication unravels as a finite process that reaches agreement among the agents; argumentation-based *protocols* are used for that purpose (Panisson et al. 2015b).

There have also been extensions of the described approach to support pondering over arguments taking into consideration information available from the agent society, the shared environment, and models of other agents engaged in argumentation-based dialogues (Panisson et al. 2018b; Melo et al. 2016). The approach has been used, for example, in mobile applications to support teamwork in healthcare scenarios involving ambient assisted living (Panisson et al. 2015a), or aiming at emerging technologies such as the Internet of Things (Panisson et al. 2018a).

Even more interestingly from the MAOP and JaCaMo point of view, the use of patterns of argumentation-based reasoning, called *argumentation schemes*, has been integrated with the organization dimension, and can be used to support domain-specific patterns of reasoning for particular MAOP applications (Panisson and Bordini 2017a). Furthermore, given the JaCaMo infrastructure, we can avoid exchanging parts of an argument when they are common knowledge in a JaCaMo multi-agent system (Panisson and Bordini 2017b).

All the work mentioned was put together as one of the first complete frameworks for practical argumentation based on multi-agent oriented programming, that is to say using environment and organization abstractions to support the argumentation process conducted by the agents.

11.3 MAOP and Software Engineering

In this book we introduced multi-agent systems as a paradigm for engineering complex software systems. In literature, this is the perspective adopted—among the others—by *agent-oriented software engineering* (AOSE) (Jennings 2000), which is about applying an agent-oriented approach to the wide spectrum of aspects that concern software engineering, from methodologies to architectures and technologies. In that view, in this section we discuss the impact that MAOP could bring to software engineering by both discussing in more detail the main principles of AOSE in dealing with complex software systems and, contextually, how MAOP could be a valuable tool to implement these principles.

Following Jennings (2000), we first consider the main well-known techniques adopted in software engineering for dealing with (complex) software systems, and then we discuss how multi-agent systems and MAOP support them. Booch et al. (2007) summarizes such techniques in

- *Decomposition*—dividing large problems into smaller, more manageable chunks, to be dealt with in relative isolation. This is useful to limit the designer's scope—at any given time only a part of the problem needs to be considered.
- *Abstraction*—defining simplified models of the system both to emphasize some of the details or properties and to suppress some others. This is useful too to confine the designer's scope, focusing on salient aspects at the expense of less relevant details.
- *Organization*—identifying and managing the interrelationships between the various problem-solving components. This helps designers tackling complexity both by enabling components to be grouped together and treated as a higher-level unit of analysis and by providing a means of describing the high-level relationships between various units.

It is useful here to consider also what it means that the system is *complex*, that is, what challenges these techniques have to tackle. Following Simon (1996), two important aspects that often characterize complex artificial systems are

- *Hierarchy*—complex software systems are often a composition of interrelated subsystems, each of which is in turn hierarchic in structure.
- *Interaction*—interactions in complex software systems typically concern two different levels: among subsystems and within subsystems, with different nonfunctional properties. In particular, the interactions within subsystems are both more frequent and more predictable, whereas the interactions among subsystems are more uncoupled. Accordingly, often subsystems can be treated almost as if they were independent of one another, but not totally independent. Simon refers to this kind of system as *nearly decomposable*.

Given these techniques and the challenges, we now consider how multi-agent systems and MAOP would provide an effective support for applying such software engineering techniques in the context of complex software systems.

Decomposition, Abstraction, and Organization with MAS and MAOP

The benefits of an agent-oriented approach to the process of engineering complex systems can be summarized in three main points (the interested reader can find in literature (e.g., Jennings [2000]; Wooldridge and Ciancarini [2001]) a comprehensive discussion about this theme).

The decomposition style promoted by MAS is natural for systems that are necessarily distributed or that have multiple loci of control. These characteristics are often found in real-world complex systems ("real systems have no top" [Meyer 1997]).

First, decentralization reduces control complexity, resulting in a lower degree of coupling between components. Second, decentralization of the control leads to have responsive systems in spite of their complexity (distribution), because agents as autonomous entities can decide which actions to do given their local situation, eventually interacting with the other components (agents) only when needed, that is, in the case of interactions among subsystems. Nevertheless, decentralization of control calls for considering interaction and coordination as a primary aspects of the system engineering. In this case, the flexibility of the interaction model is very important as soon as the inherent complexity makes it difficult or impossible to predict or analyze all the possible interaction at design time. Agent orientation promotes a design thinking in which components themselves are endowed with the ability to make decisions about the nature and scope of their interactions at runtime.

MAOP allows bringing this decomposition style from the design level to the programming level, and runtime as well. In addition, it enables further refining of decomposition by separating autonomous (agents) from nonautonomous (artifacts) components, which, in turn, enables realizing a better *separation of concerns*.

Moving to the abstraction point, the most powerful abstractions in designing software minimize the semantic distance between the units of analysis that are intuitively used to conceptualize the problem and the constructs present in the solution paradigm (Jennings 2000). This viewpoint is supported also by modern approaches to software engineering such as Domain-Driven Design (Evans 2003). In the case of complex systems, the problem to be characterized often consists of subsystems, subsystem components, interactions and organizational relationships. In this, the strong degree of correspondence between the notions of subsystems and agent organizations, and subsystem components and agents, make the agent-oriented mind-set a natural means of modeling complex systems. In this, the interplay between the subsystems and between their constituent components is most naturally viewed in terms of high-level social interactions, where, for example, agents cooperate to achieve common objectives, coordinate their actions, or negotiate to resolve conflicts.

MAOP helps keep this level of abstraction alive at programming and runtime, by means of first-class programming abstractions along multiple dimensions, from the agent dimension to the organization dimension.

Finally, about the organization point, complex systems typically involve changing webs of relationships between their various components (Jennings 2000). In that, agent-oriented approaches provide a rich set of structures for explicitly representing and managing organizations and organizational relationships (e.g., roles, norms, and social laws), as well as for modeling collective structures themselves (such as teams, joint intentions, and so forth), providing a flexible management of changing dynamically such organizational structures and relationships, including computational mechanisms for flexibly forming, maintaining, and disbanding organizations. In MAOP this is a main aspect, directly supported by first-class programming abstractions at the organizational level, both at the structural level—by means of structural abstractions such as roles and groups—and at the functional level—by means of functional abstractions such as schemes and missions, and at the normative level as well. An important point in this support is the connection at the programming level between the organization and the agent dimensions on one hand and between the organization and the environment dimensions on the other hand, so that well-defined semantic relationships are defined between concepts (e.g., goals in the agent and organization dimensions, acting on some operation in the environment counting as achieving goals in the organization).

Bringing Knowledge and Social Levels to Software Engineering

Overall, agent-oriented approaches and MAOP bring to software engineering what in literature have been called *knowledge level* and *social level*. The knowledge level, introduced by Newell (1982), accounts for describing (understanding, modeling, and designing) a system at a level that abstracts from the concrete structures and processes used to implement it or that defines operationally its behavior. A system is described and understood instead in terms of its goal(s) and knowledge available to achieve them, and assuming a *principle of rationality* as behavioral law, modeling its decision making. Figure 11.2 shows how a system can be described using a knowledge level. Conceptually, the knowledge is constructed on top of the symbolic level, which instead focuses on representations, data structures, and processes (rather than knowledge). In turn, the symbolic level can be layered upon a circuit level. The social level sits on top of the knowledge level because it enables us to model, study, and design the overall behavior of a complex system and its key conceptual structures without the need to delve into the knowledge details of the individual subsystems and their interactions.

The power of the knowledge and social level idea is that it allows us to rethink to the main aspects that concern the process of engineering—analysis, design, validation, and so forth—at a level of abstraction that is both domain-driven—because knowledge is typically at the domain level—and human-oriented—because a system is expressed in terms of the goals and (hopefully) rationality that human designers put into it. MAOP supports the knowledge level by keeping this level of abstraction also to develop as well as run the system.

These two levels provide a design space for identifying strategies to deal with two main challenges that are of concern in considering the engineering of modern software systems embedding AI technologies. The first one is about the struggle

Element	Description	Knowledge level	Social level
System	Entity to be described	Agent	Organization
Components	The system's primitive elements	Goals, actions	Agents, interactions, dependencies, organization relationships
Compositional law	How the components are assembled	Various	Roles, groups, missions, schemes
Behavior law	How the system's behavior depends upon its composition and components	Agent rationality	Organization rationality
Medium	The elements to be processed to obtain the desired behavior	Knowledge	Norms, means of influencing others, means of changing organizations

Figure 11.2
Knowledge and social levels, after Jennings (2000).

between flexibility and unpredictability. As we have said in previous chapters, both are the effect of (1) agent autonomy, and (2) interactions among agents, which may lead to *emergent behavior* at the MAS level. On the one hand, both autonomy and emergent behavior are important to make self-adaptive systems (Cheng et al. 2009) that flexibly adapt to unpredictable changes in the environment without the need for human intervention. On the other hand, that flexibility should not be at the expense of the capability to verify and validate software systems, to guarantee properties, and to have full control over the system. The availability of first-class design and programming abstraction as offered by the MAOP approach both at the knowledge and social levels makes it possible to balance flexibility and unpredictability—including the possibility to realize forms of *adjustable autonomy* (Scerri et al. 2002), in which agents may dynamically vary their own autonomy, transferring decision-making control to other entities, such as humans, or changing and adapting the way they are regulated (e.g., reorganizing) or the environment in which they are situated (e.g., deploying new artifacts). The second one is about *explainability* of systems, so that they should endow mechanisms to make understandable to human experts all aspects concerning their decisions and behavior in general.

The multidimensional approach provided by MAOP allows shaping such strategies by exploiting agents, organizations, and also the environment as first-class abstractions, in particular the power of mediated interaction (discussed in chapter 7) and of agents being able to reason about their organization (discussed in chapter 9).

Rethinking Methodologies and Tools

The availability of proper tools and methodologies are fundamental to support in practice such a perspective based on a knowledge and social levels on software engineering. In literature, several agent-oriented methodologies have been introduced in the context of AOSE; main examples are Prometheus (Padgham and Winikoff 2003) and Gaia (Wooldridge et al. 2000); the interested reader can refer to surveys such as those by Sturm and Shehory (2014); Iglesias et al. (1999); Cossentino et al. (2007) and books such as those by Henderson-Sellers and Giorgini (2005); Sterling and Taveter (2009). In several methodologies, environment and organizational concepts are used mainly in the early stages to clarify the problem to be solved. Along the process, these concepts are analyzed and, in the implementation phase, they usually disappear and are replaced by agent concepts, mainly because the considered programming tools support only the agent dimension. For instance, Prometheus uses the organizational concept of role to describe part of the agent behavior in the analysis phase. However, the program produced in the end of the process is, for instance, a set of agents. Role, in this example, is not a first-class entity in all the methodology phases. A work that initially deals with this issue is Prometheus AEOlus (Uez and Hübner 2014). It departs from Prometheus and changes some phases to produce code for tools like JaCaMo in the end of the development process.

From a programming point of view, MAOP fosters thinking of novel kinds of IDEs (Integrated Development Environments), including new perspectives for debuggers and profilers, making it possible to inspect, query, and analyze the behavior of the systems at both the knowledge and the social levels. Examples of work in literature that go in this direction are those by Hindriks (2012); Winikoff (2017), which rethink of debugging for agent-oriented programming as an explanation process driven by *why* questions. JaCaMo provides some support for that view by allowing the user to inspect at runtime agents in terms of their beliefs and intentions; workspaces in terms of artifacts and their observable properties; and organizations in terms of groups, schemes, and players that are part of the metamodel.

However, at the time of the writing of this book, we are still far from having tools that fully exploit in practice the power of the knowledge and social levels applied to software engineering. We have still a great space of opportunities for research and development toward tools that will be broadly adopted by the academy and the industry.

11.4 The Road Ahead

In "The Continuing Quest for Abstraction," Henry Lieberman, from MIT, talking about programming paradigms, object-oriented programming in particular, said

> The history of Object-Oriented Programming can be interpreted as a continuing quest to capture the notion of abstraction—to create computational artifacts that represent the essential nature of a situation, and to ignore irrelevant details.
>
> Lieberman (2006)

Multi-agent oriented programming contributes to the *continuing quest of abstraction* by introducing new first-class programming concepts that capture important aspects that characterize modern complex software systems and "situations." Every paradigm has some inspiring reference background metaphor or context—the computer machine for imperative programming, mathematics for functional programming, and the world of objects for object-oriented programming. MAOP inspiration comes from the world of humans, how they interact with each other and with artifacts, and how they organize themselves in their environment. As shown in this book, this level of abstraction triggers new ways to understand the engineering of complex software systems, as well as new ways to conceive a fruitful integration between software engineering and artificial intelligence, under the knowledge and social levels.

In fact, future application scenarios call for a wider integration perspective, beyond software engineering and artificial intelligence. In socio-technical and cyber-physical systems we see a stronger blending of the digital and the physical world, of the digital world and people, societies, institutions, and laws. This blending calls for more pervasive and novel forms of interaction and cooperation between *humans* and machines, and—in the MAOP perspective—between humans and systems engineered as organizations of intelligent agents. This direction can be recognized in research visions such as the human-agent collectives (Jennings et al. 2014) and mirror worlds (Gelernter 1991; Ricci et al. 2019), and in the human-in-the-loop perspectives (Ricci et al. 2015).

This means finding a place for humans in the knowledge and social levels picture, so that systems can be conceived as organizations of agents *and* humans as well, where the behavioral laws accounts for theories of (organization) rationality including both agents and humans. In the continuing quest of abstraction, *humane* aspects may deserve a first-class modeling and implementation. This appears an essential step for the design and development of forthcoming open, autonomous, smart socio-technical and cyber-physical societies that need to explicitly represent and operationalize high-level human principles concerning ethics, values, and law. We hope MAOP will continue to develop in that direction.

We hope this book has helped convey the idea that MAOP is an important step in the quest for abstraction and toward a future of humans-in-the-loop systems as well as humane computational systems, in which ethical artificial intelligence techniques are seamlessly integrated into the practical development of complex systems built for the social good.

Solutions to Exercises

Here we present and discuss the answers for some selected exercises. You can download the code for these answers and the answers for the other exercises from the book website

http://jacamo.sourceforge.net/book

Exercise 3.1

a) To add the word “Wonderful” in the “Hello World” message, we need to initially create a new agent `john` to handle the management of this new word. Its plans are stored in the file called `john.asl`).

   The application file for the solution based on agent *communication* is as follows:

```
mas mag_hw {
    agent bob { // file bob.asl is used
      goals: say("Hello")
    }
    agent alice
    agent john
}
```

   The code of the agent `bob` is modified to request `john` to print its word:

```
+!say(M) <- .print(M);
            .send(john,achieve,say("Wonderful")).
```

   The code of the agent `john` is very similar, requesting `alice` to print its word:

```
+!say(M) <- .print(M);
            .send(alice,achieve,say("World")).
```

   Finally, the code of `alice` does not change:

```
+!say(M) <- .print(M).
```

Programming this solution required to reprogram the agents so that they properly coordinate the execution of the shared task. We needed to change the receiver of the messages (to indicate the next agent to handle the next step of the global task) and the word they have to print.

In the application file for the *coordination using the environment*, we keep the `Blackboard` artifact and create the new agent `john` and make it focus on the artifact:

```
mas sit_hw {
  agent bob {
    join: room               // bob joins workspace toolbox
    goals: say("Hello")
  }

  agent alice {
    join: room        // alice also joins workspace toolbox
    focus: room.board        // and focus on artifact board
  }

  agent john {
    join: room        // alice also joins workspace toolbox
    focus: room.board        // and focus on artifact board
  }

  workspace room {          // creates the workspace toolbox
    artifact board: tools.Blackboard // with artifact board
  }
}
```

The code of the agent `bob` is not changed:

```
+!say(M) <- writeMsg(M).
```

The code of the agent `john` is new. It consists of one plan triggered by the perception of `lastMsg("Hello")`:

```
+lastMsg("Hello") <- writeMsg("Wonderful").
```

The code of the agent `alice` has been changed to consider the new triggering event `lastMsg("Wonderful")`:

```
+lastMsg("Wonderful") <- writeMsg("World!").
```

Programming this solution required reprogramming the agents only regarding the word they wait for and print. Different from the coordination by communication solution, we do not need to include in the agent programs who is the next agent in the task.

Finally, for the *coordination by organization*, we change the organization specification by creating a new mission responsible for printing the new word. Note the goal `show_w3`, the mission `mission3`, and the norm `norm3` in the definition of the social scheme and of the normative specification in this new organization specification (the only change in the structural specification consists in fixing the cardinality of the role `greeter` to 3):

```
<functional-specification>
  <scheme id="hw_choreography">
    <goal id="show_message">
      <plan operator="sequence">
        <goal id="show_w1"/>
        <goal id="show_w3"/>
        <goal id="show_w2"/>
      </plan>
    </goal>

    <mission id="mission1"  min="1" max="1"> <goal id="show_w1"/> </mission>
    <mission id="mission2"  min="1" max="1"> <goal id="show_w2"/> </mission>
    <mission id="mission3"  min="1" max="1"> <goal id="show_w3"/> </mission>
  </scheme>
</functional-specification>

<normative-specification>
  <norm id="norm1"  type="permission" role="greeter" mission="mission1"/>
  <norm id="norm2"  type="permission" role="greeter" mission="mission2"/>
  <norm id="norm3"  type="permission" role="greeter" mission="mission3"/>
</normative-specification>
```

The creation of agent `john` in the application file is similar to the other agents:

```
   ...

   agent john : hwa.asl {
     focus: room.board
     roles: greeter in ghw
     beliefs: my_mission(mission3)
   }
   ...
```

In the common code for all agents (file `hwa.asl`), we add a new plan for handling the new goal:

```
...
+!show_w3  <- !say("Wonderful").
...
```

The addition of a new word required us to change the organization specification and to create a new agent able to handle the allocation of mission `mission3`.

b) To print the message in reverse order, producing the solution based on communication requires changing the code of all agents, revising the receiver of the messages and the word they have to print. In the version of the coordination using the environment, we need to revise the trigger event and printed word in the plans of all agents. In the version based on the organization we need to change the order of the goals in the social scheme, no need to change the code of the agents, because the coordination is (explicitly) coded outside the agents.

c) To print the words in parallel, again, in solutions based on communication and environment, we have to revise the code of all the agents. In the version based on the organization, we simply have to replace `sequence` by `parallel` in the definition of the social scheme.

Exercise 4.1 The agent `bob` programmed in the file `bob.asl` has one belief corresponding to one of its preferred cake: `new_cake("Biscuit Cake")`. The first of its five plans implements a reactive behavior creating a new goal to have that cake. The two following plans implement the proactive behavior, decomposing the goal of having a cake into two subgoals: `buy` or `make` the cake. These two behaviors take place in different circumstances, as written in the context part (introduced by the `':'` separator) of the agent plan.

```
new_cake("Biscuit Cake"). // to simulate new cake advert
have(money).    // I initially believe I have some money

+new_cake(X) <- !have(cake(X)).      // reactive behavior

                                   // proactive behavior:
 // two possible plans to acheive the goal have(cake(_))
+!have(cake(X)) : have(money) <- !buy(X).
+!have(cake(X)) : have(flour) & have(salt) & have(sugar) &
                  have(vanilla) & have(oven)
  <- !make(X).

+!buy(X)  <- .print("Buying ",X).
+!make(X) <- .print("Making ",X).
```

Exercise 4.2 The rule to infer whether the agent has all the required conditions to bake a cake at home can be as follows:

```
have_all_to_bake :- have(flour) & have(salt) & have(sugar) &
                    have(vanilla) & have(oven).
```

and thus the second plan to achieve the goal `have(cake(X))` can be simplified to

```
+!have(cake(X)) : have_all_to_bake  <- !make(X).
```

Exercise 5.1

a) The `Calculator` artifact introduced in section 5.2 is extended to implement the new operation `sum` accessing the value of the observable property `lastResult`:

```
package tools;
import cartago.*;

public class Calculator extends Artifact {

  void init() {
    defineObsProperty("lastResult",0.0);
  }

  @OPERATION void sum(double a, OpFeedbackParam<Double> result){
    ObsProperty p = getObsProperty("lastResult");
    double res = a + p.doubleValue();
    p.updateValue(res);
    result.set(res);
  }
}
```

b) The new artifact `CalculatorB` extends the above artifact `Calculator` (use of inheritance) and implements the two new operations `storeResult` and `recall`:

```
package tools;
import cartago.*;

public class CalculatorB extends Calculator {

    double mem = 0;

    @OPERATION void storeResult() {
        mem = getObsProperty("lastResult").doubleValue();
    }

    @OPERATION void recall() {
        getObsProperty("lastResult").updateValue(mem);
    }
}
```

An example of agent testing this artifact is

```
!test. // inicial goal

+!test
  <- sum(10.1,S); sum(S,R);
     storeResult;
     sum(1000,_);
     recall.

+lastResult(S) <- .print("Sum is now ",S).

{ include("$jacamoJar/templates/common-cartago.asl") }
```

The following application file makes this agent use the new artifact:

```
mas calc {
  agent bobB {
    focus: w.calculator
  }

  workspace w {
    artifact calculator: tools.CalculatorB
  }
}
```

The result of the execution is

```
Sum is now 0
Sum is now 10.1
Sum is now 20.2
Sum is now 1020.2
Sum is now 20.2
```

Exercise 5.2 The usage interface of the `SharedDictionary` artifact is composed of one operation `put` to add new information items in the dictionary, and one operation `get` to retrieve an information (use of `OpFeedbackParam`) given a key. This artifact does not provide any observable property.

```
package tools;
import cartago.*;
import jason.asSyntax.*;
import jason.asSyntax.parser.*;
import java.util.*;

public class Dictionary extends Artifact {

  Map<String,Object> dic = new HashMap<>();

  @OPERATION void put(String k, Object v) {
    dic.put(k,v);
  }

  @OPERATION void get(String k, OpFeedbackParam<Term> r) {
    try {
      r.set(ASSyntax.parseTerm(dic.get(k).toString()));
    } catch (ParseException e) {
      failed("object "+dic.get(k)+
                  " can not be parsed as a Jason term!");
    }
  }
}
```

Exercise 6.1 The blindly committed pattern for the goal `buy` can be implemented as follows:

```
+!buy(X) : have(X). // goal achieved already, nothing to do
+!buy(X) <- goto(market); buy(X). // try to achieve the goal (subject to failure)
-!buy(X) <- !buy(X). // keep trying
```

A single-minded committed agent has an extra plan to drop the goal:

```
+!buy(X) : have(X).
+!buy(X) <- goto(market); buy(X).
-!buy(X) <- !buy(X).

-open(shop) <- .fail_goal(buy(_)). // in case I do not believe there
                 // is an open shopping, drop the goal with failure
```

Exercise 7.1 The solution to this exercise is programmed in three agents: `agentA`, `agentB` (based on the `agent.asl` program), and `agentC` programmed in `agentC.asl`. The three agents work all together in the workspace `playground` where they share the artifacts `left` and `right` based on the `Table` artifact template placed in the package `tools`. The application file for this solution is

```
mas ping_pong {
  agent agentA : agent.asl {
    focus: playground.left
           playground.right
    goals: start
  }

  agent agentB : agent.asl {
    focus: playground.left
           playground.right
  }

  agent agentC {
    focus: playground.left
           playground.right
  }

  workspace playground {
    artifact left:  tools.Table
    artifact right: tools.Table
  }
}
```

Artifacts `left` and `right` are created from the same class `Table.java`:

```
package tools;
import cartago.*;
import jason.asSyntax.*;

public class Table extends Artifact {
  @OPERATION void play() {
    // getOpUserName() returns the name of the agent
    // executing this operation
    signal("played", ASSyntax.createAtom(getOpUserName()));
    // the type of the argument is Jason Atom
  }
}
```

Agents `agentA` and `agentB` share the same code:

```
+!start <- play.
+played(A)
  :  not .my_name(A) // it is my turn
  <- .print("Agent ",A," has played");
     .wait(1000);
     play.
{ include("$jacamoJar/templates/common-cartago.asl") }
```

Finally, `agentC` program is

```
c(0). // counter as belief
+played(_) : c(X) <- -+c(X+1); .print("ping ",X).
{ include("$jacamoJar/templates/common-cartago.asl") }
```

The result of the execution is

```
[agentB] Agent agentA has played
[agentC] ping 0
[agentA] Agent agentB has played
[agentC] ping 1
[agentB] Agent agentA has played
[agentC] ping 2
[agentA] Agent agentB has played
[agentC] ping 3
[agentB] Agent agentA has played
[agentC] ping 4
```

Exercise 7.2 The implementation in `agentC` to send “stop” to the others is as follows:

```
c(0). // counter as belief
+played(_) : c(10) <- .broadcast(tell,stop). // *** new plan
+played(_) : c(X)  <- -+c(X+1); .print("ping ",X).
{ include("$jacamoJar/templates/common-cartago.asl") }
```

The implementation in agents `agentA` and `agentB` can be done in several ways. The first solution is to constrain the reaction to event `played/1` by making the plan applicable only if the agent does not have the belief `stop` (which is told by `agentC`):

```
+!start <- play.
+played(A)
  :  not .my_name(A) &
     not stop    // *** new condition
  <- play.
{ include("$jacamoJar/templates/common-cartago.asl") }
```

This solution only avoids the application of the plan for the event `+played(...)`. The intentions previously created are not affected by the `agentC` message. In that case, it would be preferable to drop those intentions using the `.drop_intention` internal action. However, this internal action `.drop_intention` can be used only for goals and the current plan is data oriented (triggered by creation of a new belief)! We need thus to refactor the code of `agentA` and `agentB` to be goal oriented (that could thus be dropped):

```
1  !play.                                    // new inicial goal
2  +!start <- play.                                // initial play
3
4  +!play
5    <-                                        // wait other play
6       .wait( last_played(A) & not .my_name(A));
7       .wait(1000);
8       play;                                           // action
9       !play.                 // continue with the goal to play
10 +stop <- .drop_intention(play).       // drop goal contition
11
12 +played(A) // store the signal as a belief (used in line 6)
13   <- .print("Agent ",A," has played");
14      -+last_played(A).
15
16 { include("$jacamoJar/templates/common-cartago.asl") }
```

Note that the `.wait` in line 6 waits for a belief; however, the artifact produces a signal, then the last plan (line 12–14) is required. This last plan could be removed if the artifact had `played/1` as an observable property instead of a signal.

Exercise 8.1 A possible organization specification (written in the `wp-os.xml` file) for structuring and coordinating the process undertaken by assistants agents to support the writing of a paper is as follows. The *structural specification* has one group `wpgroup` where the `writer` and `editor` roles are connected by "acquaintance," "authority," and "communication" links (note the use of the factorization of the "communication" link on the `author` role that is inherited by the `writer` and `editor` roles). We can notice that agents are allowed to play `editor` and `writer` at the same time (`compatibility` relation between these two roles in the scope of this group). The *functional specification* abstracts the global process of writing a paper via the definition of a plan in the social scheme `writePaperSch` decomposing goals into sequence and/or parallel subgoals. The goals of this plan are distributed into three missions: `mCollaborator`, `mManager`, `mBib`. Finally, the *normative specification* assigns to roles of the structural specification the obligations and permission to commit to the missions defined in the functional specification.

```
<?xml version="1.0" encoding="UTF-8"?>

<?xml-stylesheet href="http://moise.sourceforge.net/xml/os.xsl"
                 type="text/xsl" ?>

<organisational-specification
```

```
id="wp"
os-version="0.8"

xmlns='http://moise.sourceforge.net/os'
xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
xsi:schemaLocation='http://moise.sourceforge.net/os
                    http://moise.sourceforge.net/xml/os.xsd' >

<structural-specification>
  <role-definitions>
    <role id="author" />
    <role id="writer"> <extends role="author"/> </role>
    <role id="editor"> <extends role="author"/> </role>
  </role-definitions>

  <group-specification id="wpgroup" >
    <roles>
        <role id="writer" min="1" max="5" />
        <role id="editor" min="1" max="1" />
    </roles>
    <links>
      <link from="writer" type="acquaintance"  to="editor" scope="intra-group" />
      <link from="editor" type="authority"     to="writer" scope="intra-group" />
      <link from="author" type="communication" to="author" scope="intra-group" />
    </links>

    <formation-constraints>
        <compatibility from="editor" to="writer" type="compatibility"
                       scope="intra-group" bi-dir="true"/>
    </formation-constraints>
  </group-specification>
</structural-specification>

<functional-specification>
  <scheme id="writePaperSch" >

    <goal id="wp" ttf="5 seconds">
      <plan operator="sequence" >
        <goal id="fdv" ds="First Draft Version">
          <plan operator="sequence">
            <goal id="wtitle"     ttf="1 day" ds="Write a title"/>
            <goal id="wabs"       ttf="1 day" ds="Write an abstract"/>
            <goal id="wsectitles" ttf="1 day" ds="Write the sections' title" />
          </plan>
        </goal>
        <goal id="sv" ds="Submission Version">
          <plan operator="sequence">
            <goal id="wsecs"  ttf="7 days" ds="Write sections"/>
            <goal id="finish" ds="Finish paper">
              <plan operator="parallel">
                <goal id="wconc"  ttf="1 day"
                      ds="Write a conclusion"/>
                <goal id="wrefs"  ttf="1 hour"
                      ds="Complete references and link them to text"/>
              </plan>
            </goal>
          </plan>
        </goal>
      </plan>
    </goal>

    <mission id="mCollaborator" min="1" max="5">
      <goal id="wsecs"/>
    </mission>
```

```
      <mission id="mManager" min="1" max="1">
        <goal id="wabs"/>
        <goal id="wp"/>
        <goal id="wtitle"/>
        <goal id="wconc"/>
        <goal id="wsectitles"/>
      </mission>

      <mission id="mBib" min="1" max="1">
        <goal id="wrefs"/>
        <preferred mission="mCollaborator"/>
        <preferred mission="mManager"/>
      </mission>
    </scheme>
  </functional-specification>

  <normative-specification>
    <norm id = "n1" role="editor" type="permission" mission="mManager" />
    <norm id = "n2" role="writer" type="obligation" mission="mBib"       />
    <norm id = "n3" role="writer" type="obligation" mission="mCollaborator" />
  </normative-specification>
</organisational-specification>
```

Considering three assistant agents `bob`, `alice`, and `carol`, for instance, the application file initiates the organization entity by creating the group entity `paper_group` in which agent `bob` plays the role `editor`, `alice` and `carol` the role `writer`. This group is responsible for realizing the social scheme entity `s1` corresponding to the `writePaperSch`.

```
mas writing_paper {

    agent bob
    agent alice
    agent carol

    organisation opaper: wp-os.xml {
        group paper_group: wpgroup {
            responsible-for: s1
            players: bob editor
                     alice writer
                     carol writer
        }
        scheme s1: writePaperSch
    }
}
```

These three agents have plans and beliefs that make them obedient to the obligations that are issued by the organization entity during the execution of the global process.

Exercise 9.1 To add a new agent playing room controller, we can simply add a new `agent` entry in the application file, making this new agent focus on the `hvac` artifact and adopts the role `controller` in the group entity `r1`:

```
agent second_rc : room_controller.asl {
  focus: room.hvac
  roles: controller in smart_house_org.r1
}
```

When we run the application, we notice the following error messages:

```
[OrgArt] normative failure:
            fail(role_cardinality(controller,r1,2,1))
[second_rc] Error with initial role
            role(smart_house_org,"local",r1,controller)
```

These messages indicate that some agent is trying to violate a role cardinality constraint. To change the role cardinality so that two agents can play the role `controller` in the same group entity, we have to change the `max` attribute in the role definition in the group of the structural specification:

```
...
<role id="controller" min="1" max="2" />
...
```

Of course, it only solves the normative failure. Other problems will appear since the system is not prepared to run with two agents playing the role `controller` (e.g., coordination between these two agents). Indeed, the role cardinality fixed to 1 was there to ensure that there will be only one room controller and thus prevent this kind of problem.

Exercise 9.2 As the code is currently written in the application file, the multi-agent system will run normally even if the `controller` cardinality is set to $< 0, 1 >$. As written in the application file, there is an agent who adopts this role and thus everything runs fine. However, if we are running a system where it is possible that no agent adopts the role, the minimum cardinality is satisfied and thus the group entity can be considered well formed. It means that the social scheme entity can be set under the responsibility of the group and then the execution can be launched. On the contrary, if `min=1` and no agent plays that role, the group will not be well formed and then cannot be responsible for schemes, and the scheme will not start its execution. So, by stating `min=1`, we guarantee that the scheme starts its execution with a room controller agent engaged.

Exercise 9.3

a) Although the agent did not send its vote to achieve the goal `ballot`, the system will run normally considering that the agent has fulfilled its obligation for this goal! The reason is that goals are considered fulfilled if the agent finishes a plan for it. See the research corner on page 140 for further details.

b) Although the agent actually votes by the action `vote`, the obligation for goal `ballot` is never fulfilled! Again, the goal is fulfilled if the agent finishes the plan and, in this case, obviously, it never finishes!

Exercise 9.4 We can simply add the following plan to print out fulfilled obligations:

```
+oblFulfilled(O)
  <- .print(O," is fulfilled").
```

Unification can be used for a proper printing:

```
+oblFulfilled(obligation(Who,Condition,What,When))
  <- .print(Who," has fulfilled ",What).
```

References

Acay, Daghan L., Liz Sonenberg, Alessandro Ricci, and Philippe Pasquier. 2009. How situated is your agent? A cognitive perspective. In *Programming Multi-Agent Systems, 6th International Workshop, ProMAS 2008, Estoril, Portugal, May 13, 2008. Revised invited and selected papers*, eds. Koen V. Hindriks, Alexander Pokahr, and Sebastian Sardiña. Vol. 5442 of *LNCS*, 136–151. Springer. https://doi.org/10.1007/978-3-642-03278-3_9.

Airiau, Stéphane, Lin Padgham, Sebastian Sardiña, and Sandip Sen. 2009. Enhancing the adaptation of BDI agents using learning techniques. *International Journal of Agent Technologies and Systems* 1 (2): 1–18. https://doi.org/10.4018/jats.2009040101.

Aldewereld, Huib, Olivier Boissier, Virginia Dignum, Pablo Noriega, and Julian Padget, eds. 2016. *Social coordination frameworks for social technical systems*. Vol. 30 of *Law, governance and technology series*. Springer. https://doi.org/10.1007/978-3-319-33570-4.

Argente, Estefania, Olivier Boissier, Carlos Carrascosa, Nicoletta Fornara, Peter McBurney, Pablo Noriega, Alessandro Ricci, Jordi Sabater q. Mir, Michael Ignaz Schumacher, Charalampos Tampitsikas, Kuldar Taveter, Giuseppe Vizzari, and George A. Vouros. 2013. The role of the environment in agreement technologies. *Artificial Intelligence Review* 39 (1): 21–38. https://doi.org/10.1007/s10462-012-9388-1.

Austin, John Langshaw. 1962. *How to do things with words*. Clarendon Press.

Baldoni, Matteo, Cristina Baroglio, Federico Capuzzimati, and Roberto Micalizio. 2016. Commitment-based agent interaction in JaCaMo+. *Fundamenta Informaticae* 21: 1001–1030. https://doi.org/10.3233/FI-2015-0000.

Balke, Tina, Célia da Costa Pereira, Frank Dignum, Emiliano Lorini, Antonino Rotolo, Wamberto Vasconcelos, and Serena Villata. 2013. Norms in MAS: Definitions and related concepts. In *Normative multi-agent systems*, eds. Giulia Andrighetto, Guido Governatori, Pablo Noriega, and Leendert W. N. van der Torre. Vol. 4 of *Dagstuhl follow-ups*, 1–31. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. https://doi.org/10.4230/DFU.Vol4.12111.1.

Behrens, Tristan M., Mehdi Dastani, Jürgen Dix, Jomi Fred Hübner, Michael Köster, Peter Novák, and Federico Schlesinger. 2012. The multi-agent programming contest. *AI Magazine* 33 (4): 111–113. http://www.aaai.org/ojs/index.php/aimagazine/article/view/2439.

Behrens, Tristan M., Koen V. Hindriks, and Jürgen Dix. 2011. Towards an environment interface standard for agent platforms. *Annals of Mathematics and Artificial Intelligence* 61 (4): 261–295. https://doi.org/10.1007/s10472-010-9215-9.

Bellifemine, Fabio Luigi, Giovanni Caire, and Dominic Greenwood. 2007. *Developing multi-agent systems with JADE*. *Wiley series in agent technology*. John Wiley & Sons.

Bellman, Richard. 1957. A Markovian decision process. *Journal of Mathematics and Mechanics* 6 (5): 679–684.

Berners-Lee, Tim, James Hendler, and Ora Lassila. 2001. The semantic web. *Scientific American* 284 (5): 34–43. http://www.sciam.com/article.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21.

Bernoux, P. 1985. *La sociologie des organisations*, 3ème ed. Seuil.

Boella, Guido, and Leendert van der Torre. 2006. Constitutive norms in the design of normative multiagent systems. In *Computational logic in multi-agent systems (CLIMA VI)*, eds. Francesca Toni and Paolo Torroni. Vol. 3900 of *LNCS*, 303–319. Springer. https://doi.org/10.1007/11750734_17.

Boella, Guido, and Leendert W. N. van der Torre. 2004. Regulative and constitutive norms in normative multiagent systems. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Ninth International Conference (KR2004)*, 255–266.

Boissier, Olivier, Rafael Bordini, Jomi Fred Hübner, Alessandro Ricci, and Andrea Santi. 2013. Multi-agent oriented programming with JaCaMo. *Science of Computer Programming* 78 (6): 747–761. https://doi.org/10.1016/j.scico.2011.10.004.

Boissier, Olivier, Rafael H. Bordini, Jomi F. Hübner, and Alessandro Ricci. 2019. Dimensions in programming multi-agent systems. *The Knowledge Engineering Review* 34. https://doi.org/10.1017/S026988891800005X.

Bond, Alan H. 1990. A computational model for organizations of cooperating intelligent agents. In *Proceedings of the Conference on Office Information Systems (COIS90)*.

Booch, Grady, Robert Maksimchuk, Michael Engle, Bobbi Young, Jim Conallen, and Kelli Houston. 2007. *Object-oriented analysis and design with applications*, 3rd ed. Addison-Wesley Professional.

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

Bordini, Rafael H., Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni, eds. 2005. *Multi-agent programming: Languages, platforms and applications*. *Multiagent systems, artificial societies, and simulated organizations*. Springer.

Bordini, Rafael H., Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni, eds. 2009. *Multi-agent programming: Languages, tools and applications*. Springer.

Bordini, Rafael H., Jomi Fred Hübner, and Michael Wooldridge. 2007. *Programming multi-agent systems in AgentSpeak using Jason*. John Wiley & Sons.

Bordini, Rafael H., Amal El Fallah Sghrouchni, Koen Hindriks, Brian Logan, and Alessandro Ricci. 2019. Agent programming in the cognitive era. *Journal of Autonomous Agents and Multi-agent Systems (forthcoming)*.

Bosello, Michael, and Alessandro Ricci. 2019. From programming agents to educating agents: A Jason-based framework for integrating learning in the development of cognitive agents. In *Engineering Multi-Agent Systems - 7th International Workshop, EMAS 2019, Montreal, May 11–12*.

Bratman, M. 1987. *Intention, plans, and practical reason*. Harvard University Press.

Bratman, Michael E., David J. Israel, and Martha E. Pollack. 1988. Plans and resource-bounded practical reasoning. *Computational Intelligence* 4: 349–355. https://doi.org/10.1111/j.1467-8640.1988.tb00284.x.

Bretier, Philippe, and M. David Sadek. 1996. A rational agent as the kernel of a cooperative spoken dialogue system: Implementing a logical theory of interaction. In *Intelligent Agents III, Agent Theories, Architectures, and Languages, ECAI '96 Workshop ATAL, Budapest, Hungary, August 12–13, 1996, Proceedings*, eds. Jörg P. Müller, Michael Wooldridge, and Nicholas R. Jennings. Vol. 1193 of *LNCS*, 189–203. Springer. https://doi.org/10.1007/BFb0013586.

Broersen, Jan, and Leendert van der Torre. 2012. Ten problems of deontic logic and normative reasoning in computer science, eds. Nick Bezhanishvili and Valentin Goranko, 55–88. Springer. https://doi.org/10.1007/978-3-642-31485-8_2.

Broersen, Jan M., Stephen Cranefield, Yehia Elrakaiby, Dov M. Gabbay, Davide Grossi, Emiliano Lorini, Xavier Parent, Leendert W. N. van der Torre, Luca Tummolini, Paolo Turrini, and François Schwarzentruber. 2013. Normative reasoning and consequence. In *Normative Multi-Agent Systems*, 33–70. https://doi.org/10.4230/DFU.Vol4.12111.33.

Bromuri, S., and K. Stathis. 2008. Situating cognitive agents in GOLEM. In *Engineering Environment-Mediated Multi-Agent Systems*, eds. D. Weyns, S. Brueckner, and Y. Demazeau. Vol. 5049 of *LNCS*, 115–134. Springer.

Busetta, Paolo, Nicholas Howden, Ralph Rönnquist, and Andrew Hodgson. 1999. Structuring BDI agents in functional clusters. In *Intelligent Agents VI, Agent Theories, Architectures, and Languages, 6th International Workshop (ATAL '99), Orlando, Florida, USA, July 15–17, 1999, Proceedings*, eds. Nicholas R. Jennings and Yves Lespérance. Vol. 1757 of *LNCS*, 277–289. Springer. https://doi.org/10.1007/10719619_21.

Cardoso, Rafael C., and Rafael H. Bordini. 2017. A modular framework for decentralised multi-agent planning. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*. *AAMAS 17*, 1487–1489. International Foundation for Autonomous Agents and Multiagent Systems.

Cardoso, Rafael C., and Rafael H. Bordini. 2019. Decentralised planning for multi-agent programming platforms. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS '19, Montreal, Canada, May 13–17, 2019*, eds. Edith Elkind, Manuela Veloso, Noa Agmon, and Matthew E. Taylor, 799–818. http://dl.acm.org/citation.cfm?id=3331771.

Chalupsky, Hans, Yolanda Gil, Craig A. Knoblock, Kristina Lerman, Jean Oh, David V. Pynadath, Thomas A. Russ, and Milind Tambe. 2001. Electric elves: Applying agent technology to support human organizations. In *Proceedings of the Thirteenth Conference on Innovative Applications of Artificial Intelligence Conference*, 51–58. AAAI Press. http://dl.acm.org/citation.cfm?id=645453.652996.

Cheng, Betty H., Rogério Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. 2009. *Software engineering for self-adaptive systems*, eds. Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, 1–26. Springer. https://doi.org/10.1007/978-3-642-02161-9_1. Chap. Software Engineering for Self-Adaptive Systems: A Research Roadmap.

Ciancarini, Paolo. 1996. Coordination models and languages as software integrators. *ACM Computing Surveys* 28 (2): 300–302. https://doi.org/10.1145/234528.234732.

Ciortea, Andrei, Olivier Boissier, and Alessandro Ricci. 2019. Engineering world-wide multi-agent systems with hypermedia. In *Engineering Multi-Agent Systems - 6th International Workshop, EMAS 2018, Stockholm, Sweden, July 14–15, 2018, revised selected papers*, eds. Danny Weyns, Viviana Mascardi, and Alessandro Ricci. Vol. 11375 of *LNCS*, 285–301. Springer. https://doi.org/10.1007/978-3-030-25693-7_15.

Ciortea, Andrei, Simon Mayer, Fabien L. Gandon, Olivier Boissier, Alessandro Ricci, and Antoine Zimmermann. 2019. A decade in hindsight: The missing bridge between multi-agent systems and the world wide web. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS'19, Montreal, Canada, May 13–17, 2019*, eds. Edith Elkind, Manuela Veloso, Noa Agmon, and Matthew E. Taylor, 1659–1663. http://dl.acm.org/citation.cfm?id=3331893.

Ciortea, Andrei, Simon Mayer, and Florian Michahelles. 2018. Repurposing manufacturing lines on the fly with multi-agent systems for the web of things. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2018, Stockholm, Sweden, July 10–15, 2018*, eds. Elisabeth André, Sven Koenig, Mehdi Dastani, and Gita Sukthankar, 813–822. International Foundation for Autonomous Agents and Multiagent Systems. http://dl.acm.org/citation.cfm?id=3237504.

Cohen, Philip R., and Hector J. Levesque. 1990. Intention is choice with commitment. *Artificial Intelligence* 42 (2-3): 213–261. https://doi.org/10.1016/0004-3702(90)90055-5.

Collier, R., S. Russell, and D. Lillis. 2015. Exploring AOP from an OOP perspective. In *Proceedings of the 5th International Workshop on Programming based on Actors, Agents and Decentralized Control (held at SPLASH 2014)*.

Corkill, Daniel D., and Victor R. Lesser. 1983. The use of meta-level control for coordination in distributed problem solving network. In *Proceedings of the 8th International Joint Conference on Artificial Intelligence (IJCAI'83)*, ed. Alan Bundy, 748–756. William Kaufmann.

Cossentino, Massimo, Salvatore Gaglio, Alfredo Garro, and Valeria Seidita. 2007. Method fragments for agent design methodologies: From standardisation to research. *International Journal of Agent-Oriented Software Engineering* 1 (1): 91–121. https://doi.org/10.1504/IJAOSE.2007.013266.

Coutinho, Luciano R., Jaime S. Sichman, and Olivier Boissier. 2009. Modelling dimensions for agent organizations. In *Handbook of research on multi-agent systems: Semantics and dynamics of organizational models*, 18–50. IGI Global.

Criado, Natalia, Estefania Argente, and Vicente Botti. 2011. THOMAS: An agent platform for supporting normative multi-agent systems. *Journal of Logic and Computation* 23 (2): 309–333. https://doi.org/10.1093/logcom/exr025.

Croatti, Angelo, Sara Montagna, Alessandro Ricci, Emiliano Gamberini, Vittorio Albarello, and Vanni Agnoletti. 2018. BDI personal medical assistant agents: The case of trauma tracking and alerting. *Artificial Intelligence in Medicine*. https://doi.org/10.1016/j.artmed.2018.12.002.

Dastani, Mehdi. 2008. 2apl: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems* 16 (3): 214–248. https://doi.org/10.1007/s10458-008-9036-y.

Dastani, Mehdi, Nick Tinnemeier, and John-Jules CH. Meyer. 2009. A programming language for normative multi-agent systems. In *Multi-agent systems: semantics and dynamics of organizational models*, ed. Virginia Dignum. Information Science Reference.

de Brito, Maiquel, Jomi Fred Hübner, and Olivier Boissier. 2017. Architecture of an institutional platform for multi-agent systems. In *PRIMA 2017: Principles and Practice of Multi-Agent Systems - 20th International Conference, Nice, France, October 30 – November 3, 2017, Proceedings*, eds. Bo An, Ana L. C. Bazzan, João Leite, Serena Villata, and Leendert W. N. van der Torre. Vol. 10621 of *LNCS*, 313–329. Springer. https://doi.org/10.1007/978-3-319-69131-2_19.

de Brito, Maiquel, Jomi Fred Hübner, and Olivier Boissier. 2018. Situated artificial institutions: stability, consistency, and flexibility in the regulation of agent societies. *Autonomous Agents and Multi-Agent Systems* 32 (2): 219–251. https://doi.org/10.1007/s10458-017-9379-3.

Demazeau, Yves. 1995. From interactions to collective behaviour in agent-based systems. In *Proceedings of the 1st. European Conference on Cognitive Science*, 117–132.

Demazeau, Yves, and Antônio Carlos da Rocha Costa. 1996. Populations and organizations in open multi-agent systems. In *PDAI 96 - 1st National Symposium on Parallel and Distributed AI*.

Dennett, Daniel C. 1987. *The intentional stance*. MIT Press.

Dignum, Virginia. 2009. *Handbook of research on multi-agent systems: Semantics and dynamics of organizational models: Semantics and dynamics of organizational models*. IGI Global.

Dignum, Virginia, Javier Vazquez-Salceda, and Frank Dignum. 2004. OMNI: Introducing social structure, norms and ontologies into agent organizations. In *Proceedings of the Programming Multi-Agent Systems (ProMAS 2004)*, eds. Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni. Vol. 3346 of *LNAI*. Springer.

Drogoul, Alexis, Bruno Corbara, and Steffen Lalande. 1995. MANTA: New experimental results on the emergence of (artificial) ant societies. In *Artificial Societies: the Computer Simulation of Social Life*, eds. Nigel Gilbert and Rosaria Conte, 119–221. UCL Press.

Esteva, Marc, Juan A. Rodriguez-Aguiar, Carles Sierra, Pere Garcia, and Josep L. Arcos. 2001. On the formal specification of electronic institutions. In *Proceedings of the Agent-mediated Electronic Commerce*, eds. Frank Dignum and Carles Sierra. Vol. 1191 of *LNAI*, 126–147. Springer.

Esteva, Marc, Juan A. Rodríguez-Aguilar, Bruno Rosell, and Josep L. Arcos. 2004. AMELI: An agent-based middleware for electronic institutions. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2004)*, eds. Nicholas R. Jennings, Carles Sierra, Liz Sonenberg, and Milind Tambe, 236–243. ACM.

Evans. 2003. *Domain-driven design: Tackling complexity in the heart of software*. Addison-Wesley Longman.

Ferber, Jacques. 1999. *Multi-agent systems: An introduction to distributed artificial intelligence*, 1st ed. Addison-Wesley Longman.

Ferber, Jacques, and Olivier Gutknecht. 1998. A meta-model for the analysis and design of organizations in multi-agents systems. In *Proceedings of the 3rd International Conference on Multi-Agent Systems (ICMAS 98)*, ed. Yves Demazeau, 128–135. IEEE Press.

Finin, Tim, Richard Fritzson, Don McKay, and Robin McEntire. 1994. KQML as an agent communication language. In *Proc. of the Third International Conference on Information and Knowledge Management*. *CIKM '94*, 456–463. ACM. https://doi.org/10.1145/191246.191322.

Fisher, Michael. 1993. Concurrent MetateM - A language for modelling reactive systems. In *PARLE '93, Parallel Architectures and Languages Europe, 5th International PARLE Conference, 1993, Proc.*, eds. Arndt Bode, Mike Reeve, and Gottfried Wolf. Vol. 694 of *LNCS*, 185–196. Springer. https://doi.org/10.1007/3-540-56891-3_15.

Foundation for Intelligent Physical Agents. 2000. *FIPA ACL message structure specification*. FIPA. http://www.fipa.org.

Fox, Mark S. 1981. An organizational view of distributed systems. *IEEE Transactions on Systems, Man, and Cybernetics* 11 (1): 70–80.

Freitas, Artur, Alison R. Panisson, Lucas Hilgert, Felipe Meneguzzi, Renata Vieira, and Rafael H. Bordini. 2015. Integrating ontologies with multi-agent systems through cartago artifacts. In *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology, WI-IAT 2015, Singapore, December 6–9, 2015 - Volume II*, 143–150. https://doi.org/10.1109/WI-IAT.2015.116.

Gasser, Les. 2001. Perspectives on organizations in multi-agent systems. In *Multi-agents systems and applications*, 1–16. Springer.

Gasser, Les, Nicholas F. Rouquette, Randall W. Hill, and John Lieb. 1989. Representing and using organizational knowledge in distributed AI systems. In *Distributed artificial intelligence*, eds. Les Gasser and Michael N. Huhns, Vol. 2, 55–79. Morgan Kaufmann. Chap. 3.

Gelernter, David. 1991. *Mirror worlds or the day software puts the universe in a shoebox: How will it happen and what it will mean*. Oxford University Press.

Georgeff, Michael P., and François Felix Ingrand. 1989. Decision-making in an embedded reasoning system. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence. Detroit, MI, USA, August 1989*, ed. N. S. Sridharan, 972–978. Morgan Kaufmann. http://ijcai.org/Proceedings/89-2/Papers/020.pdf.

Georgeff, Michael P., and Amy L. Lansky. 1987. Reactive reasoning and planning. In *Proceedings of the 6th National Conference on Artificial Intelligence. Seattle, WA, USA, July 1987.*, eds. Kenneth D. Forbus and Howard E. Shrobe, 677–682. Morgan Kaufmann. http://www.aaai.org/Library/AAAI/1987/aaai87-121.php.

Geraci, Anne, Freny Katki, Louise McMonegal, Bennett Meyer, John Lane, Paul Wilson, Jane Radatz, Mary Yee, Hugh Porteous, and Fredrick Springsteel. 1991. *IEEE standard computer dictionary: Compilation of IEEE standard computer glossaries*. IEEE Press.

Ghallab, Malik, Dana S. Nau, and Paolo Traverso. 2016. *Automated planning and acting*. Cambridge University Press.

Ghose, Aditya, Nir Oren, Pankaj Telang, and John Thangarajah, eds. 2015. Coordination, organizations, institutions, and norms in agent systems X. Vol. 9372 of *LNAI*. Springer. https://doi.org/10.1007/978-3-319-25420-3.

Governatori, Guido, Michael J. Maher, Grigoris Antoniou, and David Billington. 2004. Argumentation semantics for defeasible logic. *Journal of Logic and Computation* 14 (5): 675–702.

Guerra-Hernández, Alejandro, Amal El Fallah-Seghrouchni, and Henry Soldano. 2004. Learning in BDI multi-agent systems. In *Computational Logic in Multi-Agent Systems, 4th International Workshop, CLIMA IV, Fort Lauderdale, FL, USA, January 6–7, 2004. Revised selected and invited papers*, eds. Jürgen Dix and João Alexandre Leite. Vol. 3259 of *LNCS*, 218–233. Springer. https://doi.org/10.1007/978-3-540-30200-1_12.

Gutknecht, Olivier, and Jacques Ferber. 2000. The MadKit agent platform architecture. In *Agents Workshop on Infrastructure for Multi-Agent Systems*, 48–55.

Hannoun, Mahdi, Olivier Boissier, Jaime Simão Sichman, and Claudette Sayettat. 2000. MOISE: an organizational model for multi-agent systems. In *Advances in Artificial Intelligence, International Joint Conference, 7th Ibero-American Conference on AI, 15th Brazilian Symposium on AI, IBERAMIA-SBIA 2000, Atibaia, SP, Brazil, November 19–22, 2000, Proceedings*, eds. Maria Carolina Monard and Jaime Simão Sichman. Vol. 1952 of *LNCS*, 156–165. Springer. https://doi.org/10.1007/3-540-44399-1_17.

Henderson-Sellers, Brian, and Paolo Giorgini, eds. 2005. *Agent-oriented methodologies*. IGI Global.

Hendler, James. 2001. Agents and the semantic web. *IEEE Intelligent Systems* 16 (2): 30–37. https://doi.org/10.1109/5254.920597.

Herzig, Andreas, Laurent Perrussel, and Zhanhao Xiao. 2016. On hierarchical task networks. In *Logics in Artificial Intelligence - 15th European Conference, JELIA 2016, Larnaca, Cyprus, November 9–11, 2016, Proceedings*, eds. Loizos Michael and Antonis C. Kakas. Vol. 10021 of *LNCS*, 551–557. https://doi.org/10.1007/978-3-319-48758-8_38.

Hewitt, Carl, and Peter De Jong. 1984. Open systems. In *On conceptual modelling*, 147–164. Springer.

Hindriks, Koen V. 2009. Programming rational agents in GOAL. In *Multi-Agent Programming*, eds. Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni. *Multiagent systems, artificial societies, and simulated organizations*, 119–157. Springer.

Hindriks, Koen V.. 2012. Debugging is explaining. In *Proc. of PRIMA 2012, Int. Conf. on Principles and Practice of Multi-Agent Systems*. Vol. 7455 of *LNCS*, 31–45. Springer. https://doi.org/10.1007/978-3-642-32729-2_3.

Hindriks, Koen V., Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. 1997. Formal semantics for an abstract agent programming language. In *Intelligent Agents IV, Agent Theories, Architectures, and Languages, 4th International Workshop, ATAL '97, Providence, Rhode Island, USA, 1997, Proc.*, eds. Munindar P. Singh, Anand S. Rao, and Michael J. Wooldridge. Vol. 1365 of *LNCS*, 215–229. Springer. https://doi.org/10.1007/BFb0026761.

Hindriks, Koen V., and Jürgen Dix. 2014. GOAL: A multi-agent programming language applied to an exploration game. In *Agent-Oriented Software Engineering - Reflections on Architectures, Methodologies, Languages, and Frameworks*, eds. Onn Shehory and Arnon Sturm, 235–258. Springer.

Hübner, Jomi Fred, Olivier Boissier, and Rafael H Bordini. 2011. A normative programming language for multi-agent organisations. *Annals of Mathematics and Artificial Intelligence* 62 (1-2): 27–53. https://doi.org/10.1007/s10472-011-9251-0.

Hübner, Jomi Fred, Olivier Boissier, Rosine Kitio, and Alessandro Ricci. 2010. Instrumenting multi-agent organisations with organisational artifacts and agents: Giving the organisational power back to the agents. *Journal of Autonomous Agents and Multi-Agent Systems* 20 (3): 369–400. https://doi.org/10.1007/s10458-009-9084-y.

Hübner, Jomi Fred, Jaime Simão Sichman, and Olivier Boissier. 2002. A model for the structural, functional, and deontic specification of organizations in multiagent systems. In *Proceedings of the 16th Brazilian Symposium on Artificial Intelligence (SBIA'02)*, eds. Guilherme Bittencourt and Geber L. Ramalho. Vol. 2507 of *LNAI*, 118–128. https://doi.org/10.1007/3-540-36127-8_12.

Hübner, Jomi Fred, Jaime Simão Sichman, and Olivier Boissier. 2004. Using the Moise+ for a cooperative framework of MAS reorganisation. In *Brazilian Symposium on Artificial Intelligence*, 506–515. Springer.

Hübner, Jomi Fred, Jaime Simão Sichman, and Olivier Boissier. 2007. Developing organised multiagent systems using the MOISE+ model: programming issues at the system and agent levels. *International Journal of Agent-Oriented Software Engineering* 1 (3-4): 370–395.

Huhns, Michael N., and Munindar P. Singh. 2005. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing* 9 (1): 75–81. https://doi.org/10.1109/MIC.2005.21.

Iglesias, Carlos A., Mercedes Garijo, and José C. González. 1999. A survey of agent-oriented methodologies. In *Proceedings of the 5th International Workshop on Intelligent Agents V, Agent Theories, Architectures, and Languages*, eds. Jörg P. Müller, Anand S. Rao, and Munindar P. Singh. *ATAL'98*, 317–330. Springer.

Jennings, Nicholas R. 2000. On agent-based software engineering. *Artificial Intelligence* 117 (2): 277–296. https://doi.org/10.1016/S0004-3702(99)00107-1.

Jennings, Nicholas R. 2001. An agent-based approach for building complex software systems. *Communications of the ACM* 44 (4): 35–41. https://doi.org/10.1145/367211.367250.

Jennings, N. R., L. Moreau, D. Nicholson, S. Ramchurn, S. Roberts, T. Rodden, and A. Rogers. 2014. Human-agent collectives. *Communications of the ACM* 57 (12): 80–88. https://doi.org/10.1145/2629559.

Klapiscak, Thomas, and Rafael H. Bordini. 2009. JASDL: A practical programming approach combining agent and semantic web technologies. In *Declarative Agent Languages and Technologies VI*, eds. Matteo Baldoni, Tran Cao Son, M. Birna van Riemsdijk, and Michael Winikoff, 91–110. Springer.

Lespérance, Yves, Hector J. Levesque, Fangzhen Lin, Daniel Marcu, Raymond Reiter, and Richard B. Scherl. 1996. Foundations of a logical approach to agent programming. In *Intelligent Agents II, Agent Theories, Architectures, and Languages, IJCAI'95, Workshop ATAL, Montreal, Canada, August 19–20, 1995, Proceedings*, eds. Michael Wooldridge, Jörg P. Müller, and Milind Tambe. Vol. 1037 of *LNCS*, 331–346. Springer. https://doi.org/10.1007/3540608052_76.

Li, Cuihong, Joseph A. Giampapa, and Katia P. Sycara. 2006. Bilateral negotiation decisions with uncertain dynamic outside options. *IEEE Transactions Systems, Man, and Cybernetics, Part C* 36 (1): 31–44. https://doi.org/10.1109/TSMCC.2005.860573.

Lieberman, Henry. 2006. The continuing quest for abstraction. In *Proceedings of the 20th European Conference on Object-Oriented Programming*. *ECOOP'06*, 192–197. Springer. https://doi.org/10.1007/11785477_12.

Logan, Brian, John Thangarajah, and Neil Yorke-Smith. 2017. Progressing intention progression: A call for a goal-plan tree contest. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017, São Paulo, Brazil, May 8–12, 2017*, eds. Kate Larson, Michael Winikoff, Sanmay Das, and Edmund H. Durfee, 768–772. ACM. http://dl.acm.org/citation.cfm?id=3091234.

Maes, Pattie. 1994. Agents that reduce work and information overload. *Communications of the ACM* 37 (7): 30–40. https://doi.org/10.1145/176789.176792.

Malone, Thomas W. 1999. Tools for inventing organizations: Toward a handbook of organizational process. *Management Science* 45 (3): 425–443.

Mascardi, Viviana, Davide Ancona, Matteo Barbieri, Rafael H. Bordini, and Alessandro Ricci. 2014. CooL-AgentSpeak: Endowing AgentSpeak-DL agents with plan exchange and ontology services. *Web Intelligence and Agent Systems* 12 (1): 83–107. https://doi.org/10.3233/WIA-140287.

Mayfield, James, Yannis Labrou, and Timothy W. Finin. 1996. Evaluation of KQML as an agent communication language. In *Intelligent Agents II, Agent Theories, Architectures, and Languages, IJCAI '95, Workshop ATAL, Montreal, Canada, August 19–20, 1995, Proceedings*, eds. Michael Wooldridge, Jörg P. Müller, and Milind Tambe. Vol. 1037 of *LNCS*, 347–360. Springer. https://doi.org/10.1007/3540608052_77.

Melo, Victor S., Alison R. Panisson, and Rafael H. Bordini. 2016. Argumentation-based reasoning using preferences over sources of information: (extended abstract). In *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems, Singapore, May 9–13, 2016*, eds. Catholijn M. Jonker, Stacy Marsella, John Thangarajah, and Karl Tuyls, 1337–1338. ACM. http://dl.acm.org/citation.cfm?id=2937148.

Meyer, Bertrand. 1997. *Object-oriented software construction*, 2nd ed. Prentice-Hall.

Modi, Pragnesh Jay, Manuela Veloso, Stephen F. Smith, and Jean Oh. 2005. Cmradar: A personal assistant agent for calendar management. In *Proceedings of the 6th International Conference on Agent-Oriented Information Systems II. AOIS 04*, 169–181. Springer. https://doi.org/10.1007/11426714_12.

Morin, E. 1977. *La méthode (1) : la nature de la nature*. Points Seuil.

Morris, Edwin, Linda Levine, Craig Meyers, Pat Place, and Dan Plakosh. 2004. System of systems interoperability (SOSI): final report, Technical report, DTIC Document.

Newell, Allen. 1982. The knowledge level. *Artificial Intelligence* 18 (1): 87–127. https://doi.org/10.1016/0004-3702(82)90012-1.

Nielsen, Claus Ballegaard, Peter Gorm Larsen, John Fitzgerald, Jim Woodcock, and Jan Peleska. 2015. Systems of systems engineering: Basic concepts, model-based techniques, and research directions. *ACM Computing Surveys* 48 (2): 18–11841. https://doi.org/10.1145/2794381.

Nute, Donald. 1993. *Defeasible Prolog*. *Artificial intelligence programs*. University of Georgia.

Nute, Donald. 2001. Defeasible logic. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, 353–395. Oxford University Press.

Okuyama, Fabio Yoshimitsu, Rafael H. Bordini, and Antônio Carlos da Rocha Costa. 2013. Situated normative infrastructures: the normative object approach. *Journal of Logic and Computation* 23 (2): 397–424. https://doi.org/10.1093/logcom/exr029.

Ortiz-Hernández, Gustavo, Jomi Fred Hübner, Rafael H. Bordini, Alejandro Guerra-Hernández, Guillermo J. Hoyos-Rivera, and Nicandro Cruz-Ramírez. 2016. A namespace approach for modularity in BDI programming languages. In *Engineering Multi-Agent Systems - 4th International Workshop, EMAS 2016, Singapore, Singapore, May 9–10, 2016. Revised selected and invited papers*, eds. Matteo Baldoni, Jörg P. Müller, Ingrid Nunes, and Rym Zalila q. Wenkstern. Vol. 1 0093 of *LNCS*, 117–135. https://doi.org/10.1007/978-3-319-50983-9_7.

Ossowski, Sascha. 2012. *Agreement technologies*, Vol. 8. Springer.

Padgham, Lin, and Michael Winikoff. 2003. Prometheus: A methodology for developing intelligent agents. In *Proc. of the 3rd International Conference on Agent-oriented Software Engineering III. AOSE'02*, 174–185. Springer. http://dl.acm.org/citation.cfm?id=1754726.1754744.

Padgham, Lin, and Michael Winikoff. 2004. *Developing intelligent agent systems: A practical guide*. John Wiley & Sons.

Panisson, Alison R., Asad Ali, Peter McBurney, and Rafael H. Bordini. 2018a. Argumentation schemes for data access control. In *Computational Models of Argument - Proceedings of COMMA 2018, Warsaw, Poland, September 12–14, 2018*, eds. Sanjay Modgil, Katarzyna Budzynska, and John Lawrence. Vol. 305 of *Frontiers in artificial intelligence and applications*, 361–368. IOS Press. https://doi.org/10.3233/978-1-61499-906-5-361.

Panisson, Alison R., and Rafael H. Bordini. 2016. Knowledge representation for argumentation in agent-oriented programming languages. In *5th Brazilian Conference on Intelligent Systems, BRACIS 2016, Recife, Brazil, October 9–12, 2016*, 13–18. IEEE Computer Society. https://doi.org/10.1109/BRACIS.2016.014. SBC.

Panisson, Alison R., and Rafael H. Bordini. 2017a. Argumentation schemes in multi-agent systems: A social perspective. In *Engineering Multi-Agent Systems - 5th International Workshop, EMAS 2017, São Paulo, Brazil, May 8–9, 2017, revised selected papers*, eds. Kate Larson, Michael Winikoff, Sanmay Das, and Edmund H. Durfee, 92–108. ACM. https://doi.org/10.1007/978-3-319-91899-0_6.

Panisson, Alison R., and Rafael H. Bordini. 2017b. Uttering only what is needed: Enthymemes in multi-agent systems. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017, São Paulo, Brazil, May 8–12, 2017*, eds. Kate Larson, Michael Winikoff, Sanmay Das, and Edmund H. Durfee, 1670–1672. ACM. http://dl.acm.org/citation.cfm?id=3091399.

Panisson, Alison R., Artur Freitas, Daniela Schmidt, Lucas Hilgert, Felipe Meneguzzi, Renata Vieira, and Rafael H. Bordini. 2015a. Arguing About Task Reallocation Using Ontological Information in Multi-Agent Systems. In *12th International Workshop on Argumentation in Multiagent Systems (ArgMAS)*.

Panisson, Alison R., Felipe Meneguzzi, Moser Silva Fagundes, Renata Vieira, and Rafael H. Bordini. 2014. Formal semantics of speech acts for argumentative dialogues. In *International conference on Autonomous Agents and Multi-Agent Systems, AAMAS '14, Paris, France, May 5–9, 2014*, eds. Ana L. C. Bazzan, Michael N. Huhns, Alessio Lomuscio, and Paul Scerri, 1437–1438. IFAAMAS/ACM. http://dl.acm.org/citation.cfm?id=2617511.

Panisson, Alison R., Felipe Meneguzzi, Renata Vieira, and Rafael H. Bordini. 2015b. Towards practical argumentation-based dialogues in multi-agent systems. In *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology, WI-IAT 2015, Singapore, December 6–9, 2015 - Volume II*, 151–158. https://doi.org/10.1109/WI-IAT.2015.208.

Panisson, Alison R., Simon Parsons, Peter McBurney, and Rafael H. Bordini. 2018b. Choosing appropriate arguments from trustworthy sources. In *Computational Models of Argument - Proceedings of COMMA 2018, Warsaw, Poland, September 12–14, 2018*, eds. Sanjay Modgil, Katarzyna Budzynska, and John Lawrence. Vol. 305 of *Frontiers in artificial intelligence and applications*, 345–352. IOS Press. https://doi.org/10.3233/978-1-61499-906-5-345.

Pattison, H. Edward, Daniel D. Corkill, and Victor R. Lesser. 1987. Instantiating description of organizational structures. In *Distributed artificial intelligence*, ed. Michael N. Huhns, Vol. 1, 59–96. Morgan Kaufmann. Chap. 3.

Piunti, Michele, Alessandro Ricci, Olivier Boissier, and Jomi Fred Hübner. 2009. Embodying organisations in multi-agent work environments. In *Proceedings of the 2009 IEEE/WIC/ACM International Conference on Intelligent Agent Technology, IAT 2009, Milan, Italy, September 15–18 2009*, 511–518. IEEE Computer Society. https://doi.org/10.1109/WI-IAT.2009.204.

Pokahr, Alexander, Lars Braubach, and Winfried Lamersdorf. 2005. Jadex: A BDI reasoning engine. In *Multi-Agent Programming*, eds. Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni. *Multiagent systems, artificial societies, and simulated organizations*, 149–174. Springer.

Pynadath, David V., and Milind Tambe. 2003. An automated teamwork infrastructure for heterogeneous software agents and humans. *Autonomous Agents and Multi-Agent Systems* 7 (1-2): 71–100.

Ramirez, Wulfrano Arturo Luna, and Maria Fasli. 2017. Plan acquisition in a BDI agent framework through intentional learning. In *Multiagent System Technologies - 15th German Conference, MATES 2017, Leipzig, Germany, August 23–26, 2017, Proceedings*, eds. Jan Ole Berndt, Paolo Petta, and Rainer Unland. Vol. 10413 of *LNCS*, 167–186. Springer. https://doi.org/10.1007/978-3-319-64798-2_11.

Rao, Anand S. 1996. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Agents Breaking Away, 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Eindhoven, The Netherlands, 1996, Proc.*, eds. Walter Van de Velde and John W. Perram. Vol. 1038 of *LNCS*, 42–55. Springer. https://doi.org/10.1007/BFb0031845.

Ricci, Alessandro, Andrea Omicini, Mirko Viroli, Luca Gardelli, and Enrico Oliva. 2007. Cognitive stigmergy: Towards a framework based on agents and artifacts. In *Environments for multi-agent systems III*, eds. Danny Weyns, H. Van Dyke Parunak, and Fabien Michel, 124–140. Springer.

Ricci, Alessandro, Michele Piunti, L. Daghan Acay, Rafael H. Bordini, Jomi F. Hübner, and Mehdi Dastani. 2008. Integrating heterogeneous agent programming platforms within artifact-based environments. In *Proc. of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems. AAMAS 2008*, 225–232. IFAAMAS. http://dl.acm.org/citation.cfm?id=1402383.1402419.

Ricci, Alessandro, Michele Piunti, and Mirko Viroli. 2010. Environment programming in multi-agent systems: an artifact-based perspective. *Autonomous Agents and Multi-Agent Systems* 23 (2): 158–192. https://doi.org/10.1007/s10458-010-9140-7.

Ricci, Alessandro, Michele Piunti, Mirko Viroli, and Andrea Omicini. 2009. Multi-agent programming: Languages, tools and applications, eds. Amal El Fallah Seghrouchni, Jürgen Dix, Mehdi Dastani, and Rafael Bordini, 259–288. Springer. https://doi.org/10.1007/978-0-387-89299-3_8.

Ricci, Alessandro, Juan A. Rodriguez-Aguilar, Ander Pijoan, and Franco Zambonelli. 2015. Mixed environments for MAS: Bringing humans in the loop. In *Agent Environments for multi-agent systems IV*, eds. Danny Weyns and Fabien Michel, 52–60. Springer.

Ricci, Alessandro, Luca Tummolini, and Cristiano Castelfranchi. 2019. Augmented societies with mirror worlds. *AI and Society* 34 (4): 745–752. https://doi.org/10.1007/s00146-017-0788-2.

Ricci, Alessandro, Mirko Viroli, and Andrea Omicini. 2006. Programming MAS with artifacts. In *Programming Multi-Agent Systems*, eds. Rafael H. Bordini, Mehdi M. Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni, 206–221. Springer.

Rodriguez, Sebastian, Nicolas Gaud, and Stéphane Galland. 2014. SARL: A general-purpose agent-oriented programming language. In *2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT), Warsaw, Poland, August 11–14, 2014 - Volume III*, 103–110. IEEE Computer Society. https://doi.org/10.1109/WI-IAT.2014.156.

Russell, Sean Edward, Gregory M. P. O'Hare, and Rem W. Collier. 2015. Agent-oriented programming languages as a high-level abstraction facilitating the development of intelligent behaviours for component-based applications. In *PRIMA 2015: Principles and Practice of Multi-Agent Systems - 18th International Conference, Bertinoro, Italy, 2015, Proc.*, eds. Qingliang Chen, Paolo Torroni, Serena Villata, Jane Yung q. jen Hsu, and Andrea Omicini. Vol. 9387 of *LNCS*, 501–509. Springer. https://doi.org/10.1007/978-3-319-25524-8_32.

Russell, Stuart, and Peter Norvig. 2003. *Artificial intelligence, a modern approach*, 2nd ed. Prentice Hall.

Sardiña, Sebastian, and Lin Padgham. 2011. A BDI agent programming language with failure handling, declarative goals, and planning. 23 (1): 18–70. https://doi.org/10.1007/s10458-010-9130-9.

Sardiña, Sebastian, Lavindra de Silva, and Lin Padgham. 2006. Hierarchical planning in BDI agent programming languages: a formal approach. In *5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2006), Hakodate, Japan, May 8–12, 2006*, eds. Hideyuki Nakashima, Michael P. Wellman, Gerhard Weiss, and Peter Stone, 1001–1008. ACM. https://doi.org/10.1145/1160633.1160813.

Scerri, Paul, David V. Pynadath, and Milind Tambe. 2002. Towards adjustable autonomy for the real world. *Journal of Artificial Intelligence Research* 17 (1): 171–228. http://dl.acm.org/citation.cfm?id=1622810.1622816.

Searle, John. 1969. *Speech acts*. Cambridge University Press.

Searle, John. 2010. *Making the social world:the structure of human civilization*. Oxford University Press.

Searle, John R. 1997. *The construction of social reality*. Free Press.

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

Shoham, Yoav, and Kevin Leyton-Brown. 2008. *Multiagent systems: Algorithmic, game-theoretic, and logical foundations*. Cambridge University Press.

Silva, Lavindra de. 2018a. Addendum to "HTN acting: A formalism and an algorithm." abs/1806.02127. http://arxiv.org/abs/1806.02127.

Silva, Lavindra de. 2018b. HTN acting: A formalism and an algorithm. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2018, Stockholm, Sweden, July 10–15, 2018*, eds. Elisabeth André, Sven Koenig, Mehdi Dastani, and Gita Sukthankar, 363–371. International Foundation for Autonomous Agents and Multiagent Systems. http://dl.acm.org/citation.cfm?id=3237441.

Silva, Lavindra de, Sebastian Sardina, and Lin Padgham. 2009. First principles planning in BDI systems. In *8th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009), Budapest, Hungary, May 10–15, 2009, Volume 2*, eds. Carles Sierra, Cristiano Castelfranchi, Keith S. Decker, and Jaime Simão Sichman, 1105–1112. IFAAMAS. https://dl.acm.org/citation.cfm?id=1558167.

Simari, Guillermo Ricardo, and Iyad Rahwan, eds. 2009. *Argumentation in artificial intelligence*. Springer. https://doi.org/10.1007/978-0-387-98197-0.

Simon, Herbert A. 1996. *The sciences of the artificial*, 3rd ed. MIT Press.

Singh, Dhirendra, Sebastian Sardiña, and Lin Padgham. 2010a. Extending BDI plan selection to incorporate learning from experience. *Robotics and Autonomous Systems* 58 (9): 1067–1075. https://doi.org/10.1016/j.robot.2010.05.008.

Singh, Dhirendra, Sebastian Sardiña, Lin Padgham, and Stéphane Airiau. 2010b. Learning context conditions for BDI plan selection. In *9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2010), Toronto, Canada, May 10–14, 2010, Volume 1-3*, eds. Wiebe van der Hoek, Gal A. Kaminka, Yves Lespérance, Michael Luck, and Sandip Sen, 325–332. IFAAMAS. https://dl.acm.org/citation.cfm?id=1838252.

Singh, Dhirendra, Sebastian Sardiña, Lin Padgham, and Geoff James. 2011. Integrating learning into a BDI agent for environments with changing dynamics. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16–22, 2011*, ed. Toby Walsh, 2525–2530. IJCAI/AAAI. https://doi.org/10.5591/978-1-57735-516-8/IJCAI11-420. http://ijcai.org/proceedings/2011.

Singh, Munindar P. 1991. A logic of situated know-how. In *Proceedings of the 9th National Conference on Artificial Intelligence, Anaheim, CA, USA, July 14–19, 1991, Volume 1.*, eds. Thomas L. Dean and Kathleen R. McKeown, 343–348. AAAI Press / MIT Press. http://www.aaai.org/Library/AAAI/1991/aaai91-053.php.

Sterling, Leon, and Kuldar Taveter. 2009. *The art of agent-oriented modeling*. MIT Press.

Stratulat, Tiberiu, Jacques Ferber, and John Tranier. 2009. MASQ: towards an integral approach to interaction. In *AAMAS (2009)*, 813–820.

Sturm, Arnon, and Onn Shehory. 2014. The landscape of agent-oriented methodologies. In *Agent-Oriented Software Engineering*, eds. Onn Shehory and Arnon Sturm, 137–154. Springer.

Sutton, Richard S., and Andrew G. Barto. 2018. *Reinforcement learning: An introduction*. MIT Press.

Tambe, Milind. 1997. Towards flexible teamwork. *Journal of Artificial Intelligence Reseearch* 7: 83–124.

Tambe, Milind. 2008. Electric elves: What went wrong and why. *AI Magazine* 29 (2): 23–27. http://www.aaai.org/ojs/index.php/aimagazine/article/view/2123.

Theraulaz, Guy, and Eric Bonbeau. 1999. A brief history of stigmergy. *Artificial Life* 5 (2): 97–116. https://doi.org/10.1162/106454699568700.

Tolk, Andreas, and James A Muguira. 2003. The levels of conceptual interoperability model. In *Proceedings of the 2003 Fall Simulation Interoperability Workshop*, Vol. 7, 1–11. Citeseer.

Trentin, Iago Felipe, Olivier Boissier, and Fano Ramparany. 2019. Insights about user-centric contextual online adaptation of coordinated multi-agent systems in smart homes. In *Actes des 17èmes Rencontres des Jeunes Chercheurs en Intelligence Artificielle, RJCIA 2019, Toulouse, France, July 2–4, 2019.*, ed. Maxime Lefrançois, 35–42. https://hal.archives-ouvertes.fr/hal-02160421.

Uez, Daniela Maria, and Jomi F. Hübner. 2014. Environments and organizations in multi-agent systems: From modelling to code. In *Proc. 2nd International Workshop on Engineering Multi-agent Systems (EMAS @ AAMAS 2014)*, eds. Fabiano Dalpiaz, Jürgen Dix, and M. Birna van Riemsdijk. Vol. 8758 of *LNCS*, 181–203. Springer. https://doi.org/10.1007/978-3-319-14484-9_10.

Van Dyke Parunak, H. 1997. "Go to the ant": Engineering principles from natural multi-agent systems. *Annals of Operations Research* 75 (0): 69–101. https://EconPapers.repec.org/RePEc:spr:annopr:v:75:y:1997:i:0:p:69-101:10.1023/a:1018980001403.

Van-Roy, Peter, and Seif Haridi. 2004. *Concepts, techniques, and models of computer programming*. MIT Press.

Vieira, Renata, Álvaro F. Moreira, Michael J. Wooldridge, and Rafael H. Bordini. 2007. On the formal semantics of speech-act based communication in an agent-oriented programming language. *Journal of Artificial Intelligence Research* 29: 221–267. https://doi.org/10.1613/jair.2221.

Wagner, Thomas, John Phelps, Valerie Guralnik, and Ryan VanRiper. 2004. Coordinators: Coordination managers for first responders. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems. AAMAS 2004*, 1140–1147. IEEE Computer Society. https://doi.org/10.1109/AAMAS.2004.97.

Walton, Douglas, Chris Reed, and Fabrizio Macagno. 2008. *Argumentation schemes*. Cambridge University Press. http://www.cambridge.org/us/academic/subjects/philosophy/logic/argumentation-schemes.

Weiss, Gerhard, ed. 1999. *Multiagent systems: A modern approach to distributed artificial intelligence*. MIT Press.

Weyns, Danny, and Tom Holvoet. 2006. A reference architecture for situated multiagent systems. In *Environments for Multi-Agent Systems III, Third International Workshop, E4MAS 2006, Hakodate, Japan, May 8, 2006. Selected revised and invited papers*, eds. Danny Weyns, H. Van Dyke Parunak, and Fabien Michel. Vol. 4389 of *LNCS*, 1–40. Springer. https://doi.org/10.1007/978-3-540-71103-2_1.

Weyns, Danny, Andrea Omicini, and James Odell. 2007. Environment as a first class abstraction in multiagent systems. *Autonomous Agents and Multi-Agent Systems* 14 (1): 5–30. https://doi.org/10.1007/s10458-006-0012-0.

Weyns, Danny, and H. Van Dyke Parunak, eds. 2007. Special issue on environments for multi-agent systems, Vol. 14, 1–116. Springer.

Winikoff, Michael. 2005. JACK intelligent agents: An industrial strength platform. In *Multi-Agent Programming*, eds. Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni. *Multiagent systems, artificial societies, and simulated organizations*, 175–193. Springer.

Winikoff, Michael. 2017. Debugging agent programs with why?: Questions. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*. *AAMAS'17*, 251–259. International Foundation for Autonomous Agents and Multiagent Systems. http://dl.acm.org/citation.cfm?id=3091125.3091166.

Wooldridge, M. 2000. *Reasoning about rational agents*. MIT Press.

Wooldridge, Michael, and Paolo Ciancarini. 2001. Agent-oriented software engineering: The state of the art. In *First International Workshop, AOSE 2000 on Agent-oriented Software Engineering*, 1–28. Springer. http://dl.acm.org/citation.cfm?id=370834.370836.

Wooldridge, Michael, and Nicholas R. Jennings. 1995. Intelligent agents: theory and practice. *The Knowledge Engineering Review* 10 (2): 115–152. https://doi.org/10.1017/S0269888900008122.

Wooldridge, Michael, Nicholas R. Jennings, and David Kinny. 2000. The GAIA methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems* 3 (3): 285–312. https://doi.org/10.1023/A:1010071910869.

Wooldridge, Michael J. 2009. *An introduction to multiagent systems*, 2nd ed. Wiley.

Xu, Mengwei, Kim Bauters, Kevin McAreavey, and Weiru Liu. 2018. A formal approach to embedding first-principles planning in BDI agent systems. In *Scalable Uncertainty Management - 12th International Conference, SUM 2018, Milan, Italy, October 3–5, 2018, Proceedings*, eds. Davide Ciucci, Gabriella Pasi, and Barbara Vantaggi. Vol. 11142 of *LNCS*, 333–347. Springer. https://doi.org/10.1007/978-3-030-00461-3_23.

Zarafin, Alexandra-Madalina, Antoine Zimmermann, and Olivier Boissier. 2012. Integrating semantic web technologies and multi-agent systems: A semantic description of multi-agent organizations. In *Proceedings of the First International Conference on Agreement Technologies, AT 2012, Dubrovnik, Croatia, October 15–16, 2012*, eds. Sascha Ossowski, Francesca Toni, and George A. Vouros. Vol. 918 of *CEUR workshop proceedings*, 296–297. CEUR-WS.org. http://ceur-ws.org/Vol-918/111110296.pdf.

Zavoral, Filip, Jason J. Jung, and Costin Badica, eds. 2014. Intelligent distributed computing VII - proceedings of the 7th international symposium on intelligent distributed computing, IDC 2013, Prague, Czech Republic, September 2013. Vol. 511 of *Studies in computational intelligence*. Springer. https://doi.org/10.1007/978-3-319-01571-2.

Index