Organizing Agent Organizations

Syntax and Operational Semantics
of an
Organization-Oriented Programming Language

This research has been supported by NWO,
The Netherlands Organisation for Scientific Research,
CoCoMAS project, grant nr: 612.066.516.

SIKS Dissertation Series No. 2011-02
The research reported in this thesis has been carried out under the auspices of
SIKS, the Dutch Research School for Information and Knowledge Systems.

ISBN 978-90-39354896

Organizing Agent Organizations

Syntax and Operational Semantics
of an
Organization-Oriented Programming Language

Het Organiseren van Agent Organisaties

Syntax en Operationele Semantiek
van een
Organisatie-Georiënteerde Programmeertaal

(met een samenvatting in het Nederlands)

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Universiteit Utrecht
op gezag van de rector magnificus, prof. dr. J.C. Stoof, ingevolge
het besluit van het college voor promoties in het openbaar te
verdedigen op maandag 7 februari 2011 des middags te 4.15 uur

door

Nick Antonius Marinus Tinnemeier

geboren op 7 november 1980, te Enschede.

Promotor:	Prof. dr. J.-J. Ch. Meyer
Co-promotoren:	Dr. M.M. Dastani
	Dr. F.S. de Boer

Dit proefschrift werd (mede) mogelijk gemaakt met financiële steun van de Nederlandse Organisatie voor Wetenschappelijk Onderzoek (NWO)

Voor Marlies.

Omdat ik nog even in het land van de bietebauw mocht blijven.

Contents

1 Introduction **1**
1.1 Organizational Abstractions in Software Engineering 2
1.2 Agents and Multi-Agent Systems 4
1.3 Organizing Multi-Agent Systems 6
1.4 Main Motivation and Research Goal 7
1.5 Research Method and Validation 8
1.6 Chapter Overview 12
1.7 Relation with Earlier Published Work 13
1.8 Not Covered by this Thesis 14

2 Background **17**
2.1 Agents 18
2.1.1 Programming Agents 20
2.2 Multi-Agent Systems: Agents and Environments 23
2.3 Organization-Oriented Multi-Agent Systems 27
2.4 Norms 30
2.4.1 On the Logic of Norms 33
2.4.2 Towards Norms in Computational Multi-Agent Systems . . 37
2.4.3 Normative Frameworks in Multi-Agent Systems 42
2.5 Roles and Organizational Structure 49
2.5.1 The Characteristics of Roles in Computer Science 49
2.5.2 Roles in Multi-Agent Systems 50
2.6 On the Notion of Power 54
2.7 Discussion 55

3 Setting the Scene: Organizational Artifacts **59**
3.1 Conference Management System Example 60
3.2 Organizational Artifacts 61

3.3 Programming Organizational Artifacts 67
3.4 Executing Organizational Artifacts 71
3.4.1 Preliminaries 72
3.4.2 Agent Level Transitions 74
3.4.3 Artifact Level Transitions 75
3.4.4 Multi-Agent Level Transitions 79
3.4.5 Organizational Coordination Cycle 81
3.5 Discussion 84
3.6 Conclusion 86

4 Programming Regulative Norms 87
4.1 Organizational Artifacts with Norms 90
4.2 Programming the Normative Dimension 94
4.3 Executing the Normative Dimension 99
4.4 Organizational Coordination Strategies 109
4.4.1 Monitoring and Triggering Strategies 109
4.4.2 Sanctioning Strategies 113
4.5 On the Notion of Normative Conflict and Coherency 114
4.6 Conclusion 116

5 Programming Organizational Structure 117
5.1 Organizational Artifacts with Roles 120
5.1.1 Role Enactment 121
5.1.2 Position Configuration and Execution 122
5.1.3 Positions and The Normative Dimension 125
5.2 Programming Organizational Artifacts with Roles 125
5.3 Executing Organizations with Positions 131
5.3.1 Preliminaries 131
5.3.2 Executing Positions 133
5.3.3 Execution of Organizational Artifacts 135
5.4 Expressing Structural Constraints with Norms 141
5.5 Discussion 143
5.6 Conclusion 146

6 Programming Norm Change 147
6.1 Challenges and Issues for Computational Norm Change 148
6.2 Organizational Artifacts with Changing Norms 152
6.3 Programming Norm Change 154
6.3.1 Changing Norm Instances 155
6.3.2 Changing Norm Schemes 157
6.4 Operational Semantics 159
6.5 Towards a Mechanism for Avoiding Normative Conflicts 165
6.6 Related Work 170
6.7 Conclusion and Future Work 171

7 Conclusion **173**
7.1 Main Contributions 173
7.2 Ten Things to Think About Before Finishing This Thesis 176

A A Prototype Implementation **181**
A.1 General Architecture 181
A.2 Organizational Artifacts Simulated in Jess 183
A.3 Roles Simulated in Modular 2APL 187
A.4 Inspecting Organizational Artifacts 189
A.5 Conclusion and Lessons Learned 189

Bibliography **193**

Summary **207**

Samenvatting **209**

Dankwoord **211**

Curriculum Vitae **213**

SIKS Dissertation Series **215**

Chapter 1

Introduction

We do not use computers merely as smart calculators and advanced type writers anymore. We use them to find and purchase the cheapest, yet most comfortable, flight to a conference, we let them plan a route without traffic jams from Utrecht to Schiphol airport, and let them autonomously sell our ticket in an auction when we discover we exceeded our budget and refrain from attending. This development does not only make life easier, it also goes hand in hand with an increasing complexity of software systems. Much of software engineering is concerned with dealing with this ever increasing complexity, a research issue that is considered one of the foundations of the field of computing science (Wing, 2008).

Software engineering is moving away from machine-oriented views of programming towards concepts and metaphors that more closely reflect the way in which humans understand the world. Indeed, the history of software engineering is marked by the introduction of engineering abstractions which allow us to focus only on a few essential concepts at the same time by factoring out details. Well-known examples are: the abstract data types stack and queue; the concepts mailbox and semaphore from concurrency; and files, folders and registries as used by operating systems. The key idea is that by using these concepts by which we understand and structure the complex world around us we can also understand and structure (or engineer) complex software systems.

Besides aforementioned metaphorical constructs, also more high-level concepts have appeared in the area of software engineering that are based on social concepts we use for organizing our society, such as the roles we play within an organization (e.g. playing the role of student within a university) and the norms we should adhere to (e.g. handing in your homework before the deadline). The field of agent-oriented software engineering – as a relative young sub-field of software engineering and artificial intelligence – draws heavily upon metaphorical abstractions inspired by such organizational abstractions. In this thesis we are interested in organizing multi-agent systems by means of these organizational concepts. However, before we explain what an agent and a multi-agent system is and how social concepts are used to organize them, let us first briefly look at the use of organizational abstractions (norms and roles in particular) in the area of "traditional" software engineering.[1] We do so, to show that the use of roles and norms in object-orientation is differently from their use in agent-orientation.

1.1 Organizational Abstractions in Software Engineering

Within the area of computer science the concept of role has gained much attention and a myriad of different uses can be found in the area of "traditional" software engineering. The first usage of roles in computer science can be traced back at least as far as to the work of Bachman (1977) who is often seen as the one who has introduced the concept of role into data modeling. Data modeling is concerned with structuring and representing data with the purpose of storing, manipulating and retrieving it. In traditional data modeling techniques data was expressed as Pascal-like records and one could express one-to-many relations between one instance of one record type (the owner) and multiple instances of another record type (the members). The relation between a company (the owner) and the managers (the members) that work for it is an example of such a one-to-many relation. However, expressing the concept of employment modeling that, for instance, a company or a university or even another person employs a manager, technician, secretary, etc. requires a many-to-many relation involving records of different types on both the side of owner and members. This is not easily modeled in traditional data models. Bachman observed that in modeling relations between owner and member record types there have always been two roles involved, e.g. in our example the role of employer and employee. He introduced an elegant solution that essentially boils down to associating records not directly by means of one-to-many relations, but allowing multiple records (e.g. employer) to be associated via one role segment type (e.g. employment) to multiple other records (e.g. employees).

Although the relational model that does not have an explicit account of the role concept became the predominant data model, Bachman's proposal was not the last word on roles in software engineering. Quite the contrary is true. The notion of role has also appeared in the field of object-orientation. There the concept of role is basically used to address two issues:

1. An object possesses a set of attributes and methods that cannot be changed after the object is instantiated. That is to say, no additional behavior can be attached or removed to the object once it has been instantiated.

2. During the lifetime of an object different objects may be associated to it

[1]By traditional software engineering we mean the part of software engineering that does not directly use agent-oriented concepts, but of course the border between traditional software engineering and agent-oriented software engineering is not always clear.

and use its behavior in different ways. Although associated objects typically only use specific parts of the object's behavior, it is not possible to restrict the attributes and methods that are visible to other objects on the basis of their association. That is, methods and attributes can only be made private or public for all associations, but not just for one association in particular.

Roles are seen as a suitable candidate to solve these issues in object-oriented programming. One example is the work of Kristensen (1995) showing that the properties associated to the concept of role is essentially different from that of class and object in object-orientation, and cannot be simulated by standard techniques such as aggregation, association and inheritance. Therefore, he introduces roles as first-class implementation entities in object-oriented programming. Just like an object a role has methods and state, but what (amongst others) distinguishes it from an object is that unlike an object a role cannot exist without the object it is associated to (its player), and shares its identity with that of the object. Objects dynamically acquire and abandon roles, thereby dynamically extending the behaviors of the role-playing object. From the outside the role-playing object can be accessed through the roles it currently plays. The behavior that can be used by an object interacting through a player's role is restricted to the behavior offered by that role. Examples of related efforts to integrate roles in object-oriented programming are the role-object pattern (Bäumer et al., 1998) and the deployment of aspect-oriented programming to implement roles (Kendall, 1999).

Baldoni *et al.* (2005; 2007) introduce powerJava that takes a different approach to roles. Whereas many (e.g. (Kristensen, 1995; Bäumer et al., 1998; Kendall, 1999)) take a player-centered view on roles meaning that roles are directly coupled to their players and dynamically extend the behavior of the role-playing object, they take an organization-centered view. The idea is that a role describes the way objects can interact with and within an organization. In (Baldoni et al., 2007) an example is given in which a printer is seen as an organization that can be accessed through two roles, user and superuser. Objects can enact these roles providing them powers to act upon this organization. More specifically, the roles are implemented as inner classes of the class that implements the organization and empower the objects that play them, by providing methods implemented in the role, to access and change the state of the organization by means of method invocation. The methods a player can invoke are restricted to the ones the role provides. Conversely the role requires its players to possess certain capabilities by requiring certain methods in the class that implements the player, e.g. a user must provide a method for obtaining the player's name to be printed on the page headers.

Another use of roles can be found in the area of access control that is concerned with mechanisms for regulating access from users to computer-based information systems, e.g. granting a lecturer of the computer science department rights to look up records in the student administration database. Now suppose that the computer science department decides to no longer allow lecturers to access the student administration database. If we assigned the permission to access the database to each individual lecturer directly, this means we have to alter the permissions of all lecturers. To circumvent such problems, Sandhu (1996) proposed to associate permissions to roles rather than to users directly. Roles may be combined in a hierarchy where roles inherit permissions from their super-roles. Users that are authorized to be assigned a role then obtain all the permissions that are associated to that role. Changing the access permissions of lecturers then boils down to modifying only the permission associated to the (say) lecturer role.

Observe the use of permissions in RBAC. In RBAC one can only perform those operations one is permitted to. That is to say, the performance of other operations is forbidden and are made impossible by the system. These concepts clearly have a normative flavour. The use of such normative concepts is extended by Park and Sandhu (2002) who laid the basis for what they refer to as usage control. Usage control extends (amongst others) traditional access control to also include obligations. An obligation is used to describe the requirements one has to fulfill after having obtained or exercised rights on an object. For example, a requirement to share logging information with the vendor after having downloaded an application. In (Park and Sandhu, 2002), however, the concept of obligation is only discussed informally. Nothing is said about how to represent and use them in a computer system, e.g. how to verify if they are complied with and how to respond to their violation.

In the area of agents and multi-agent systems the concepts of role and norm have also deserved a lot of attention. Although there are similarities in how they are used, in the context of agency they have been used for quite distinct purposes. To understand this, we first briefly explain what agents and multi-agent system are.

1.2 Agents and Multi-Agent Systems

Agent-oriented software engineering is a promising paradigm that aims at developing complex, distributed applications, and is centered around the abstractions of agent and multi-agent system. Although the term agent is really an umbrella term for a wide variety of concepts, within the field of agent-oriented software engineering the concept of (intelligent or rational) agent is commonly used to mean:

> "a computer system that is situated in some *environment* and that is capable of *autonomous action* in this environment in order to meet its design objectives." (Wooldridge, 2001)[2].

The notion of agency derives from philosophical studies pertaining to models seeking to explain human behavior, and the approach in which agents are attributed

[2]The definition presented in (Wooldridge, 2001) is an adapted version of the definition originally presented in (Wooldridge and Jennings, 1995).

a mental state consisting of beliefs, goals and plans has gained wide acceptance in the field of multi-agent systems research. An agent's beliefs model the information the agent has abouts its world, whereas the goals describe the situation the agent wishes to achieve. Its plans describe what the agent should do to achieve its goals. In this view, an agent can thus be roughly considered the metaphorical engineering abstraction of a human being. What is seen as the defining characteristic of an agent it is its autonomy. Autonomy is an agent's capability to decide itself what to do. This implies that an autonomous agent has control over its internal state and decision making process. When mental notions are attributed to an agent, autonomy means that the agent itself decides what to belief, which goals to adopt and how to achieve them.

A multi-agent system consists of a collection of agents that interact with each other by sending communication messages and perform their actions in a shared environment. Communication between agents is typically done by means of high-level communication messages based on dedicated agent communication languages. Central to such communication languages are the so-called performative verbs, which correspond to verbs such as inform, request and promise. By means of such communication messages agents can, for example, inform another agent about some information and request or promise an agent to achieve a certain goal. It is then the agent's own decision whether to belief the information or adopt the goal. A multi-agent system can thus somehow be considered the metaphorical counterpart of a human society.

To better understand the concept of an agent and the scope of this thesis, it is helpful to understand the difference between a software agent that resides in a multi-agent system and an object as we know it from object-orientation. The most important characteristic that distinguishes agents from objects is the characteristic of autonomy. To see why an object is generally not autonomous is best explained by an example that can be found in (Wooldridge, 2001):

> Suppose we have a Java system containing two objects, o_1 and o_2, and that o_1 has a publicly available method m_1. Object o_2 can communicate with o_1 by invoking method m_1. In Java, this would mean o_2 executing an instruction that looks something like `o1.m1(arg)`, where `arg` is the argument that o_2 wants to communicate to o_1. But consider: which object makes the decision about the execution of method m_1? Is it object o_1 or object o_2? In this scenario, object o_1 has no control over the execution of m_1 the decision about whether to execute m_1 lies entirely with o_2.

This is quite different from agents that communicate by means of messages rather than method invocation, and decide themselves what to do with the content of the message. In the agent community how agents differ from objects is often captured by the popular slogan: "objects do it for free; agents do it for money."

1.3 Organizing Multi-Agent Systems

Agents participating in a multi-agent system may have been engineered by different parties with different design objectives. In a multi-agent system agents may encounter and interact with other agents having different (conflicting) goals. An illustrative example is an online marketplace on which agents interact with (unknown) parties to sell and buy their goods. Each buyer presumably wants to buy its desired goods at the lowest price possible, a goal that does not match well with other buyers who are after the same good and sellers who want to make as much money as possible. Because little can be assumed about the behavior the interacting agents will exhibit and nobody directly wrote the whole program encompassing the multi-agent system it is hard to predict the emerging behavior of the system as a whole. This makes it difficult to guarantee that the system will achieve the objectives it is designed for. One of the challenges in the design and development of such multi-agent systems is to coordinate and regulate the behavior of the individual agents to increase the likelihood that the design objectives of the system will be met.

To overcome the coordination problems that arise in the context of a multi-agent system in which different agents interact, a plethora of coordination techniques has been proposed to date. Most coordination techniques involve exogenous[3] (as opposed to endogenous) coordination media that are introduced as first-class entities that are designed, implemented and deployed separately from the agents they coordinate. Real-life examples of exogenous coordination media include blackboards for coordinating information exchange and traffic lights for regulating traffic.

Some coordination media are centered around rather low-level concepts such as action (or message) synchronization (e.g. (Omicini et al., 2004; Arbab et al., 2008; Aştefănoaei et al., 2009b)), shared data spaces (e.g. (Omicini and Zambonelli, 1999)) and resource access relations (e.g. (Cremonini et al., 2000)). Yet others focus on more high-level organizational concepts such as norms that should be followed and roles that agents can play. (Examples of these organization-oriented approaches will be presented throughout this thesis.) Organizational constructs are assumed to be useful because they accord with how we conceive and structure the world around us. In agent systems roles are particular useful to abstract away from the individuals that will play them. Getting back to the online marketplace example, agents play the roles of seller and buyer, and are expected to abide by certain norms, e.g. paying the price agreed. The focus of this dissertation is on programming these exogenous coordination media – henceforth called *organizational artifacts* – directly in terms of these organizational concepts.

Although there are similarities between organizational abstractions as we know them from "traditional" software engineering and agent-oriented software engi-

[3]The word exogenous is derived from the Greek words "exo" meaning "outside" and "gen" meaning "production".

neering, it is important to already point out why organizational abstractions as used in "traditional" software engineering cannot be directly applied in the agent world. The rationale of deploying norms and roles in agent-oriented software engineering is to regulate the behavior of agents over whom little control can be exerted. In object-orientation roles are deployed to dynamically extend an object's behavior and restrict the way other objects may interact with a role-playing object depending on the context of the interaction. Agents and objects are based on quite distinct features; objects are built of attributes and methods and interaction is primarily based on method invocation, whereas agents are autonomous entities that are built of a mental state and communicate via high-level communication languages. In usage control the use of permissions and obligations is suggested for regulating how (external) parties may access and use digital objects, but no guideline is given on how to do this. Moreover, in usage control permissions cannot be deviated from. As we shall see later on, in the agent world this is highly undesirable as it limits the agents' autonomy. Because of these important differences, for the rest of this thesis, we will focus on how organizational abstractions are used in the area of multi-agent systems, rather than "traditional" software engineering.

1.4 Main Motivation and Research Goal

To support the claim that the field of multi-agent systems in general, and agent-oriented software engineering in particular, can make a significant contribution to the development of complex, distributed software, there is a strong need for dedicated programming languages that allow us to straightforwardly build multi-agent systems directly in terms of their key concepts (Dastani et al., 2004a)[preface]. Indeed, a myriad of agent-oriented and organization-oriented programming languages have been proposed to date. Agent-oriented programming languages focus on constructs for programming individual agents such as actions, plans, goals and beliefs. Organization-oriented programming languages allow a programmer to implement a computational organization separately from the participating agents directly in terms of social constructs such as groups, shared plans, roles and norms.

Research on programming agent organizations progressed rather independently from research on programming individual agents, leaving a gap between agent programming and organization programming. In particular:

- In the organization-oriented approach the agents that will participate in the multi-agent system are intentionally treated as black-boxes, meaning that their internals are viewed as unknown. This choice is mainly justified by the fact that in a deployed multi-agent system it is assumed that the participants are not a priori known. Not making *any* assumptions about the types of agents that will participate, however, has led to a mismatch between the representation of individual agent concepts (viz. goals, beliefs and plans) and organization-oriented concepts (viz. roles and norms) (Winikoff, 2009).

- Most agent programming languages are underpinned by a formal (operational) semantics explaining the meaning of their programming constructs in an unambiguous manner. On the contrary, most approaches to constructing agent organizations lack a formal description of their constructs. This makes it difficult to thoroughly investigate the language's semantics, validate it by showing the properties it exhibits and formally compare it with other languages.

Motivated by these issues the general goal of this research is to develop an organization-oriented programming language. The proposed programming constructs for implementing agent organizations should accord with the key concepts and characteristics associated with individual agents. Moreover, the programming language should be underpinned by a formal semantics in order to unambiguously present the meaning of the proposed constructs and make claims about its main properties.

1.5 Research Method and Validation

To achieve our main research goal to try to bridge the gap between agent-oriented and organization-oriented programming we first need to know what the core ingredients of an organization-oriented programming should be, and what characteristics they should (or should not) have. Considering the large amount of different approaches and proposals that have appeared to date this is a not a trivial task. To identify the programming language's core elements and their essential characteristics we proceed as follows:

(a) By studying literature from the field of multi-agent systems we formulate the defining characteristics of our intended application area, i.e. that of agents and multi-agent systems, focusing on how to program them.

(b) To identify the key concepts dictated by the organization-oriented view we take the already existing organization-oriented implementation frameworks and design models as a starting point.

(c) Considering the defining characteristics as identified in step 1, we study the essential properties these concepts should exhibit by: 1) briefly reviewing literature from the scientific fields they originated from, e.g. philosophy and sociology, and 2) studying literature from the field of multi-agent systems in particular and from the field of computer science in general on their use as first-class engineering abstractions. The focus will be on their use in multi-agent systems development.

Comparing the output of step (c) with existing approaches enables us to support the claim that there indeed exists a gap between agent-oriented programming and organization-oriented programming, which justifies our main research goal.

Further, it leads to the identification of some of the key reasons causing this gap. Addressing each of these reasons can be considered the sub-goals whose fulfillment contributes to our overall research goal. To understand what these sub-goals are, let us make a small leap ahead and peek forward to the main causes underlying the alleged gap that will be identified in chapter 2:

1. Organizational frameworks reside in (and are even part of) the agents' environment. Most existing organizational frameworks, however, are merely used for structuring and regulating the agents' mutual interactions. However, an environment should also (amongst other) be able to encapsulate resources and provide functionality the agents may use in achieving their objectives. It is our aim to integrate these capabilities in an organizational artifact.

2. In many approaches to organizational frameworks the notion of norm is one that relates to (communicative) actions an agent should or should not perform, rather than to a declarative description of a state that should or should not be achieved. This does not accord well with the concept of goal as it is often used in agent-programming languages, viz. a high-level description of a state typically involving a complex sequence of actions to establish it. We aim to develop a solution in which the norms refer to declarative descriptions, rather than actions.

3. Because it is hard to at design-time predict the behavior the agents will exhibit, there is not one-size fits all specification of the norms. Allowing for the norms to change at run-time allows us to better cope with the actuality of the running system. However, little work has been done up to now to support the run-time modification of the norms. It is our goal to devise a mechanism that allows the norms to be changed at run-time. Because computational frameworks that address this issue are few defining the desired characteristics it should exhibit is not trivial. Moreover, such a mechanism should be carefully designed such that it does have minimal impact on the representation and meaning of the norms that are changed.

4. The concept of role can be seen as the cornerstone of an agent organization. There are many different interpretations of the concept of role, and a crisp definition is still lacking. Most organizational frameworks take a very simple view in which a role is merely a label, but other approaches have shown that a role can be a computational entity with its own state and behavior. Research from the direction of agent-oriented programming has even shown that roles may be constructed by agent-oriented concepts such that agents can reason with them. It is our goal to define the key properties a role should have. These can be easily extracted from existing literature. A more challenging problem is how to develop a solution that unites all those properties in one single proposal.

5. The structure of an organization is to a large extent encompassed by various constraints on the roles, e.g. a constraint that a role cannot be played by the same agent at the same time. In current approaches such constraints are typically expressed as hard-constraints that cannot be deviated from and which are fixed at design-time. Such an approach limits the agents' autonomy and undermines the flexibility of the organizational artifact to adapt to the dynamic interactions of the agents. It is our goal to propose a way to define these constraints as soft-constraints that can be deviated from.

6. Agents are faced with choices about what to do next, e.g. executing a plan to reach a goal or updating beliefs on the basis of incoming percepts. The best order in which agents perform such activities is application-dependent and is encoded by their deliberation cycle. An organizational framework is faced with comparable decisions about how to perform its activities, e.g. enforcing all norms versus enforcing only some in time critical systems. However, in many existing approaches the order in which the framework performs its activities is fixed. Our goal is to propose a mechanism, which we will call organizational coordination cycle, by which the different orders for executing the programming constructs can be expressed. The challenge here is that the programming constructs and in particualr their semantics should be designed such that they can be executed in various orders.

Having identified the core elements of our organizational programming language along with the essential characteristics they should posses to decrease the gap between agent-oriented and organization-oriented programming, we proceed as follows:

(d) We devise the syntax and intuitive semantics of a programming language for the key concepts as identified in step (b). Defining the syntax and intuitive semantics will be guided by the key characteristics as formulated by step (c) and well-known software engineering principles, e.g. data hiding and separation of concerns. More specifically, we will break up the programming lanuage in a set of coherent constructs each dealing with some of the above sub-goals (the relation with the chapter can be found below), viz.

    - basic constructs for programming functionality that may be exploited by agents to achieve their goals (cf. sub-goal 1) and a mechanism for expressing the organizational coordination cycle (cf. sub-goal 6).
    - constructs for expressing normative concepts in a declarative manner (cf. sub-goal 2)
    - constructs for expressing roles and constructs for expressing their structural constraints (cf. sub-goals 4 and 5).
    - constructs for programming run-time norm change (cf. sub-goal 3).

(e) We profoundly explain the meaning of the syntax by means of a Plotkin style operational semantics (Plotkin, 1981). Besides the reasons for choosing a formal approach in general, the choice for this particular type of semantics is justified by the fact that it is already close to the implementation of an interpreter. However, when implementing an interpreter for a programming language many important issues related to the semantics of the programming constructs will be lost in details of these algorithms and data structures which might even be specific to the programming language (e.g. Java or C++) the interpreter is implemented in. An additional advantage is that an operational semantics opens the way for formal verification and model checking purposes (see also (Baier and Katoen, 2008)).

(f) We use the operational semantics to formally investigate and demonstrate some of the language's elementary properties.

Formulating the key properties our language exhibits not only increases understanding, it also allows us to make claims about the "rightness" of the constructs and their interplay by formally proving that some desirable properties are exhibited. Validation of our results thus largely depends on a formal analysis. Of course, not all properties can be mathematically evaluated. This is mainly due to the fact that although syntax and semantics of a programming language are important, its raison d'être is largely determined by its pragmatics, i.e. the way in which the language is intended to be used in practice (Watt, 2004). Ideally, the pragmatics of a programming language are validated (if it can be done at all) by an empirical study on the basis of well-defined measurements (cf. (Sammet, 1971; Fenton and Pfleeger, 1998)), but this requires an existing implementation. In this research, however, we are concerned with *designing* a programming language and finding the proper programming constructs. Although the first steps have been taken to construct a prototype implementation (see appendix A) we used to experiment with and evaluate some of the language constructs, a fully-fledged implementation is not available already.

To at the very least create a better understanding of the pragmatics of the various programming constructs we propose throughout this thesis and the intuitions behind them, (besides the prototype that is described in appendix A), we rely upon an example involving the implementation of a (simplified) conference management system. This example touches upon many interesting aspects in the construction of an agent organization and reflects many of the characteristics of our problem domain (see also (Zambonelli et al., 2003)). Examples of more challenging (and perhaps more realistic) domains in which a multi-agent system approach is taken using organizational abstractions are: applications for the support of knowledge management in which different stakeholders interact and collaborate to share their knowledge within a company (Dignum, 2004); the highly-regulated area of organ and tissue allocation (Vázquez-Salceda, 2004); crisis management in which the activities of and interactions between several organizations such as police, fire and medical departments need to be coordinated in calamity situations (Quillinan et al., 2009) and; the domain of warehouse management systems in which logistical activities are organized (Hiel et al., 2010).

1.6 Chapter Overview

In this dissertation we use a bottom-up approach in constructing our programming language. After having analyzed the core ingredients for programming organizational artifacts, we will start with the elementary concepts that are needed for programming a basic organizational artifact. In each subsequent chapter we expand the set of constructs with more sophisticated organizational constructs. Each programming construct that is proposed throughout this thesis is explained by an operational semantics, cf. research step (e) of section 1.5. A concise overview of the individual chapters and their relation with the aforementioned research goals and steps from section 1.5 is given below:

Chapter 2 provides the context and background of this research. More specifically, it identifies the defining characteristics of agents and multi-agent systems and motivates why norms and roles are the key organizational concepts involved in engineering agent organizations. The emphasis of this chapter will be on programming multi-agent systems using the organizational metaphor. This chapter corresponds to research steps (a–c) and ends with the identification and refinement of our sub-goals.

Chapter 3 introduces the concept of organizational artifact that is central to this research. We explain how an organizational artifact provides functionality that can be exploited by the agents in reaching their objectives and present the syntax of the elementary programming constructs by which they are programmed. Also the concept of organizational coordination cycle is introduced by which different strategies for executing the programming constructs can be expressed. The intuitive semantics is explained by means of our running example of a conference management system. This chapter provides both the conceptual and technical basis for the rest of the thesis. This chapter addresses sub-goals 1 and 6.

Chapter 4 extends the notion of organizational artifact to include a normative dimension encompassing an explicit representation of conditional obligations and prohibitions, and sanctioning rules to punish infringements and reward obedience. Contrasting related work on organizational frameworks the norms we present refer to declartive descriptions of a sitaution that should (or should not) be established rather than an action that should or should not be performed. This chapter addresses sub-goal 2.

Chapter 5 focuses on the structure of an organizational artifact, which is largely specified in terms of roles and their mutual relationships and their constraints. We propose a view on roles in which they are computational entities with their own state and behavior. In particular, they have the ability to guide their players in interacting with the organizational artifact in a meaningful way. A role becomes the interface through which agents interact with an organizational artifact and provide functionality the agents may use in pursuit of their goals. Moreover, we investigate to what extend the constraints on roles can be expressed by the norms introduced in the previous chapter. This chapter thus mainly addresses sub-goals 4 and 5, and partially addresses sub-goal 1.

Chapter 6 proposes a mechanism to change the norms at runtime. More specifically, rule based constructs are introduced that specify when and how the norms can be changed either by the agents or the organizational artifact directly. As we shall see in this chapter, changing the norms at runtime makes it even harder to predict and avoid normative conflicts, e.g. being obliged and at the same time forbidden to view a file. In this chapter, a mechanism is investigated for avoiding normative conflicts at runtime in the context of the norm change mechanism. This chapter addresses sub-goal 3.

Chapter 7 concludes this dissertation and reflects upon to which extent we achieved our research goal. We do so by summarizing our main results and providing pointers to future research.

1.7 Relation with Earlier Published Work

The content of this dissertation is based on research we have published before in a series of papers. A list of chapters and their relation with earlier published work can be found below. In this dissertation, these earlier developed ideas are extended and integrated to form a coherent story.

Chapter 3 is based on ideas that have been presented in (Dastani et al., 2008), (Dastani et al., 2009) and (Tinnemeier et al., 2009c). These articles have formed the basis of the notion of organizational artifact as we present it in this dissertation.

Chapter 4 is an extension of work presented in (Tinnemeier et al., 2009c). The operational semantics presented in this dissertation is reworked to increase readability and make it fit with the idea of an organizational coordination cycle.

Chapter 5 is mostly based upon (Tinnemeier et al., 2009b) which presents the notion of role as it is presented in this dissertation. This chapter also includes some of the ideas about expressing organizational constraints that were presented in (Tinnemeier et al., 2009a) for a different representation of norms.

Chapter 6 is an extended version of (Tinnemeier et al., 2010). Also the operational semantics presented in this chapter is slightly reworked to fit the idea of an organizational coordination cycle. The mechanism for avoiding normative conflicts was not introduced in (Tinnemeier et al., 2010).

1.8 Not Covered by this Thesis

In this thesis we design the syntax and operational semantics of an organization-oriented programming language for programming organizational artifacts. Although a wide variety of organizational concepts has been proposed to date, the core organizational ingredients addressed in this thesis are centered around the concept of norm and role. A study of other organizational concepts is beyond the scope of this thesis. Also beyond the scope of this thesis is how the agents use and reason with the roles and norms that are provided by the organizational artifacts. This is a very important topic indeed, but this thesis will focus solely on the artifact's perspective.

An operational semantics describes the meaning the programming constructs of a programming language have in a precise manner. An operational semantics describes how a program written in the programming language is interpreted as sequences of computational steps. Such a description is very close to how a machine executes the programming constructs and is thus (at least when designed properly) already close to the implementation of an interpreter. However, having an operational semantics does not imply that implementing an interpreter is easily done. In implementing it one is, for example, faced with choices about which data structures to use and designing (efficient) algorithms for computations that in the operational semantics are succinctly described by a mathematical function. Although we have experimented with some (partial) implementations of the operational semantics (see appendix A for a discussion on this prototype implementation), how to implement an interpreter is beyond the scope of this thesis. Consequently, and as already mentioned, a discussion of the language's pragmatics is also outside the scope of this thesis.

The implementation of a multi-agent system is often preceded by a design phase in which a conceptual and technical design of the system are produced. Such a phase is typically accompanied by a modeling language and methodology. Indeed, also within the area of agent-oriented software engineering many design models and methodologies exist, some of which use organizational concepts (e.g. Gaia (Wooldridge et al., 2000) and OperA (Dignum, 2004)). Ideally, a programming language allows for a straightforward implementation of the system's design. Just like our programming language, these design models and methodologies are still under development. Although the constructs that can be found in our language are inspired by the concepts one can find in these models, an investigation of how well the programming language we propose accords with existing design methodologies is beyond the scope of this thesis.

Finally, a lot of work can be found on the formalization and operationalization of legal norms for the purpose of legal reasoning. The idea is that one tries to capture legal codes in a formal framework with the purpose of (automatically) reasoning with them. In this dissertation we are interested in how norms can be used for programming computer applications. Such programming constructs should meet people's intuitive expectations. We have no such ambitious goal as to design a language for implementing legal codes.

Chapter 2

Background

Our approach of designing and programming an organization-oriented programming language is based on ideas from research in multi-agent systems, normative systems, roles and research on programming languages for multi-agent systems. Therefore, this section introduces some of the key concepts that we use from these areas. In this thesis we devise a programming language for building organizational artifacts; software entities with the purpose of coordinating and regulating the behavior of autonomous agents that interact within a multi-agent system. The programming language we present is centered around organizational abstractions such as roles and norms. The main contributions of this chapter are as follows:

- We present a brief overview of the research areas of agents (section 2.1) and multi-agent systems (section 2.2), as agents that interact in a multi-agent system are the entities we aim to coordinate. A good understanding of these concepts is key when it comes to reducing the gap we argued to exist between agent-oriented programming languages and approaches to constructing organization-oriented multi-agent systems. Particular attention will be paid to the environment agents are deployed in, as the environment plays an important role in coordinating and regulating agents.

- We elaborate upon the coordination problems that arise in the construction of multi-agent systems and how the widely adopted organizational view on multi-agent systems can help to solve these problems (section 2.3). This motivates our choice of incorporating organization-oriented abstractions in our programming language. Moreover, it explains the role the environment plays in coordinating agents and motivates our decision of adopting an artifact based approach.

- We provide a concise overview of the concepts of norm (section 2.4) and role (section 2.5), as these are the key constituents of the organization-oriented view. Information presented in this section will be used in later chapters to devise the main characteristics these concepts ought to exhibit in our programming language. We give a brief (non-exhaustive) overview of some exemplar approaches to engineering computational, organization-oriented multi-agent systems that have appeared in literature (section 2.4 and 2.5). The aim is to give an impression of how these organization-oriented concepts are used in practice and to support our claim of the existing gap between engineering computational agents and computational agent organizations. Because the main focus of this thesis is on the concepts of role and norm, we limit the discussion of existing frameworks to these concepts only.

Because our aim is to design a programming language for implementing computational agent organizations, the emphasis of this chapter will be on implementation related issues. Risking to state the obvious, although we draw many inspiration from the field of artificial intelligence dedicated to build intelligent systems, the perspective we adopt here is mainly a software engineering perspective dedicated to designing and implementing software systems intelligently. Finally, we conclude this chapter by summarizing the main results and observations by getting back to our claim about the gap between agent-oriented implementation frameworks on the one side and organization-oriented approaches on the other.

2.1 Agents

Agents are the entities that engage in unpredictable interactions whose behaviors we aim to regulate with the goal of making the outcomes of these interactions more predictable. In the introduction we adopted Wooldridge's definition of an agent (Wooldridge, 2001), who defines it as *"an encapsulated computer system that is situated in some environment and that is capable of flexible, autonomous action in that environment in order to meet its design objectives."* Even though it clarifies the intuition of the concept, a better understanding is needed to define requirements for the organization-oriented programming language we are after. In (Wooldridge and Jennings, 1995) Wooldridge and Jennings present a (weak) notion of agency that has gained acceptance with a large group of agent researchers. According to this (weak) notion agents are characterized by being:

Autonomous. Agents are able to operate on their own without the need for human guidance while in the process of achieving their design objectives.

Social. In order to satisfy their design objectives agents typically interact and cooperate with other agents.

Pro-active. Agents are able to exhibit goal-directed behavior in order to satisfy their design objectives. This is an important aspect in distinguishing agents from traditional software: *"conventional computer software is task oriented rather than goal oriented; that is, each task (or subroutine) is executed without any memory of why it is being executed."* (Georgeff et al., 1999)

Reactive. Agents perceive their environment and adapt their behavior to changes to this environment by modifying their information and objectives accordingly.

Autonomy is commonly seen as the most important characteristic. It assigns an agent the ability to make its own decisions about, for example, which objectives to achieve first and how to achieve them. A more elaborate discussion on the notion of autonomy in the context of agency can be found in (Van der Vecht, 2009).

A stronger notion of agency that is becoming increasingly popular is the notion in which agents are attributed a mental state formed by beliefs (the possibly incorrect information about the world), desires (goals) and intentions (goals that are chosen to be actively pursued).

Assigning beliefs and desires to entities in describing and explaining their behavior is referred to as the intentional stance, as first coined by the philosopher Dennett (1987). The underlying thought is that describing the behaviour of a complex system in terms of its architecture is hardly practical. Using the intentional stance to describe an agent's behaviour in terms of the intuitive concepts belief and desire allows us to abstract from the complex (technical) inside of the system. We can predict and explain its behavior without having to reveal its insides. Based on the same ideas, Bratman (1987) developed the BDI (Beliefs, Desires and Intentions) theory of practical reasoning in which it is argued that rational behavior needs, besides beliefs and desires, also intentions, i.e. the desires we have committed to. To explain the intuition of these concepts and some other issues involved, we use an example in which we describe an agent using the BDI model. Suppose we are an agent having two different desires: to finish our dissertation within time and to visit many conferences in exotic places. Then, hopefully, we will believe that visiting conferences all the time will most likely stand in the way of finishing our dissertation. Assuming we go for having a dissertation instead of visiting conferences, we commit to the first as our intention. Having determined *what* state of affairs we want to achieve is only one part of the process. After having adopted our intention we have to figure out what we should exactly do to achieve it; we plan to read literature about the subject, publish some ground-breaking articles and finally write our dissertation. As simple as it may seem, our beliefs about the world may change while pursuing our goal, causing, for example our plan to become unachievable (the problem we are solving has already been solved twenty years ago), more attractive options to arise (we win the lottery), or worse, our intention becomes unachievable (the university goes bankrupt). Concluding, we must continuously reconsider our options and verify whether our adopted plans and intentions remain both attractive and achievable.

The process of adopting intentions based on our desires and beliefs is called *deliberation*. The second step of the process, answering the question of *how* to achieve this state of affairs, is being referred to as *means-end-reasoning* or *planning*. In literature planning and plan execution are commonly put under the umbrella of deliberation. This is not that surprising as all three steps are intimately related.

Various agent researchers recognized that the intentional stance provides us with an elegant abstraction tool for describing, explaining, and predicting the behaviour of agents in a familiar and convenient way. Hence, several proposals have been made for formalizing the intentional notions in order to specify and reason about the desired properties of agents' behavior. The most well-known examples of such a formalization are the work of Cohen and Levesque (1990), and Rao and Georgeff (1991) who capture the BDI concepts in a modal logic (Chellas, 1980).

2.1.1 Programming Agents

It has been widely recognized that the BDI model revealed itself not only as a good abstraction to explain complex systems' behavior, but also serves as a good architecture to implement agents. In the BDI architecture, plans represent the knowledge of an agent of how to achieve a certain state of affairs, and can be seen as a special kind of belief. The computational importance of plans, however, make that they are sensibly separated out as a different component (Georgeff et al., 1999). Examples of realizations of the BDI architecture are the Intelligent Resource-Bounded Machine Architecture (IRMA) (Bratman et al., 1988) developed by Bratman et al. and the Procedural Reasoning System (PRS) developed by Georgeff and Lansky (1987). These are examples of agent *architectures*, the need for dedicated *programming languages* aimed at developing agents, became apparent after the publication of Shoham's seminal paper (Shoham, 1993), in which he presents a new programming paradigm called Agent Oriented Programming (AOP). An agent oriented-language, AGENT-0, is presented in which agents can be programmed directly in terms of the mentalistic, intentional notions developed by agent theorists to represent the properties of an agent. Shoham interprets an agent as: *"...an entity whose state is viewed as consisting of mental components such as beliefs, capabilities, choices and commitments"* (Shoham, 1993).

Since the introduction of AGENT-0, a variety of BDI-style agent programming languages saw the light of day. Examples include GOAL (de Boer et al., 2007), AgentSpeak(L) (Rao, 1996) and its revival Jason (Bordini et al., 2005), the Java-based language Jadex (Pokahr et al., 2005), 3APL (short for An Abstract Agent Programming Language) (Hindriks et al., 1999) and 2APL (short for A Practical Agent Programming Language) (Dastani, 2008). Even though all languages differ in details concerning both syntax and semantics, they are roughly centered around the same concepts. Another interesting thing to remark is that all of these languages except for Jadex are underpinned by an operational semantics (Plotkin, 1981), which allows for a rigid explanation of the semantics of the syntactical constructs by which agents are programmed. This operational semantics facilitated the investigation of the relationship of some of the programming language's properties with some of the properties that are investigated by BDI-logics, cf. (Cohen and Levesque, 1990; Rao and Georgeff, 1991).

To explain the main concepts an agent-oriented programming language offers, we take 2APL as a representative. We chose 2APL, because it offers the richest set of BDI-based programming constructs that are all underpinned by an operational semantics. The idea is to create a better understanding of the concepts involved, and as we will see later on, parts of 2APL will be used in explaining our notion of roles in chapter 5. 2APL provides programming constructs to implement individual agent concepts, i.e. one can specify an agent in terms of declarative concepts such as beliefs and goals, and imperative concepts such as events and plans. In this section we briefly describe those parts of the syntax and intuitive semantics of 2APL that are relevant to this thesis. For a complete overview of the syntax and formal semantics of the 2APL programming language we refer to (Dastani, 2008).

The (possibly incorrect) information an agent has about itself and its environment is represented by its beliefs. The agents beliefs are stored in its belief base, which is implemented as a Prolog program. The situation an agent wants to realize are represented by its goals, which are stored in its goal base as conjunctions of ground atoms. A 2APL agent has only goals it does not believe to have achieved yet; any goals that become believed to be reached are automatically dropped. This particular type of goal is often referred to as "achievement goal", i.e. a declarative description of a situation the agent wants to bring about. It should be emphasized that also other types of goals exist, viz. performative goals that are a description of an action an agent wants to perform, and maintenance goals which are a description of a state an agent wants to maintain (see (van Riemsdijk et al., 2008) for a more extensive overview). To illustrate these concepts, consider an agent that is programmed to buy products in an online auction. The belief and goal base of this agent are displayed on lines 6–11 of code fragment 2.1. In this example, the agent believes its current bid to be EUR 0, the highest bid also EUR 0, and will increase its bid each round with EUR 30 to a maximum of 300.

To achieve its goals an agent needs to act. Actions that a 2APL agent can perform include actions to update its beliefs, actions to query its belief and goal base, external actions to interact with its environment (implemented as a Java program) and communicative actions to send messages to other agents. The belief base of the agent can be updated by the execution of a belief update action. Such belief updates are specified in terms of pre- and post-conditions. Intuitively, an agent can execute a belief update action if the pre-condition of the action is derivable from its belief base. After the execution of the action, the beliefs of the agent are updated such that the post-condition of the action is derivable from the agents belief base. The belief update for updating the current highest bid, for example, is specified on line 2 of code fragment 2.1.

Code fragment 2.1 2APL implementation of a buyer agent.

```
 1 BeliefUpdates:
 2   {highest(X)} UpdateHighest(H) {not highest(X), highest(H)}
 3   {true} Bought(X) {bought(X)}
 4   {permitted(X)} UpdatePermitted(P) {not permitted(X), permitted(P)}
 5
 6 Beliefs:
 7   step(30). maxBid(300). highest(0).
 8   permitted(no).
 9
10 Goals:
11   bought(bike)
12
13 PG-rules:
14   bought(X)
15   <- permitted(yes) and step(S) and maxBid(M) and highest(H) |
16   { if B(M > H + S) then
17     { @auction(bid(X, H + S), _); UpdateBid(H + S) }
18     else
19     { @auction(bid(X, 0), _) };
20     UpdatePermitted(no)
21   }
22
23 PC-rules:
24   event(endOfRound(prod(X),highest(H),perm(P),win(W)), auction)
25   <- true |
26   { if B(W = yes) then
27     { Bought(X) }
28     else
29     { UpdateHighest(H);
30       UpdatePermitted(P)
31     }
32   }
```

A 2APL agent adopts plans to achieve its goals. These plans are the recipes that describe which actions the agent should perform to reach the desired situation. In particular, plans are built of basic actions and can be composed (amongst others) by a sequence operator (i.e., ;) and a conditional choice operator. An agent possibly has more than one plan to reach its goals. Which plans are the best usually depends on the current situation. Planning goal rules are used to generate plans based on the agents goals and beliefs. Such a rule is typically of the form κ `<-` φ `|` π. The head of the rule κ is a goal expression indicating whether the agent has a certain goal. The guard φ of the rule is a belief expression indicating whether the agent has a certain belief, and the body of the rule π is the plan that can be used to achieve the goal as stated by the head. A planning goal rule can be applied if the head and guard of the rule can be entailed by the agents beliefs and goals, respectively. As an example, consider the planning goal rule of our auction agent listed on lines 14–21 of code fragment 2.1. This planning goal rule states that when the agent has a goal to have bought a product and believes to be permitted to make a new bid, it places a bid of 30 higher than the current highest bid (possibly its own bid). That is, if this new bid does not exceed the maximum of 300. Otherwise, the agent gives up by bidding 0. Note that bidding is done by performing an external action `bid` in an environment named `auction`.

To react to its dynamically changing environment, the agent is equipped with procedure call rules of the form ϕ `<-` φ `|` π allowing the agent to adopt a plan π if event or communication message ϕ is received and belief expression φ can be entailed by the agent's beliefs. Our auction agent, for example, at the end of each round receives an event with information about whether the agent won the auction, is permitted to bid and the highest bid of this round. The agent responds by updating its beliefs accordingly. Note that if the agent is told to have won the auction for product `X`, it will not pursue its goal to have bought X further.

The beliefs, goals, plans and reasoning rules form the mental states of a 2APL agent. What the agent should do with these mental attitudes is defined by means of its deliberation cycle. The deliberation cycle states which step the agent should perform next, e.g. execute an action, apply a reasoning rule or respond to external events. The deliberation cycle can thus be viewed as the meta-level control cycle of the agent program, as it determines which programming construct which we find at the object level should be executed next. It has been observed by Dastani *et al.* (2003) that there is not one single best deliberation strategy for an agent, and therefore, the agent's deliberation cycle should be programmable to flexibly cope with different situations.

2.2 Multi-Agent Systems: Agents and Environments

A multi-agent system can be defined as a collection of possibly interacting (human or software) agents that are situated in an environment and each have their own sphere of visibility and influence with respect to this environment (Jennings, 2000). This view is depicted in figure 2.1. There is, however, more to a multi-agent system than this simple definition suggests. Software engineering is facing the trend in which the whole program is no longer statically engineered by one party. Instead, a program is typically made up of different interacting entities which are each designed by different parties. Moreover, which interactions take place amongst which entities may not be known at design time. Therefore, an important factor for the successful use of multi-agent system technology as a tool to build complex software systems lies in its characteristic of "openness". Open systems are characterized by the following properties as listed by Hewitt (1986):

Concurrency. Open systems are built of multiple physically separated components. These components operate concurrently to process the large amount of information which simultaneously enters the system from outside sources.

Asynchrony. Information from the environment is usually unpredictable and may enter the system at any time. This typically results in an asynchronous interaction of the system with its environment. Moreover, the components of the system itself are possibly deployed on different platforms (distributed)

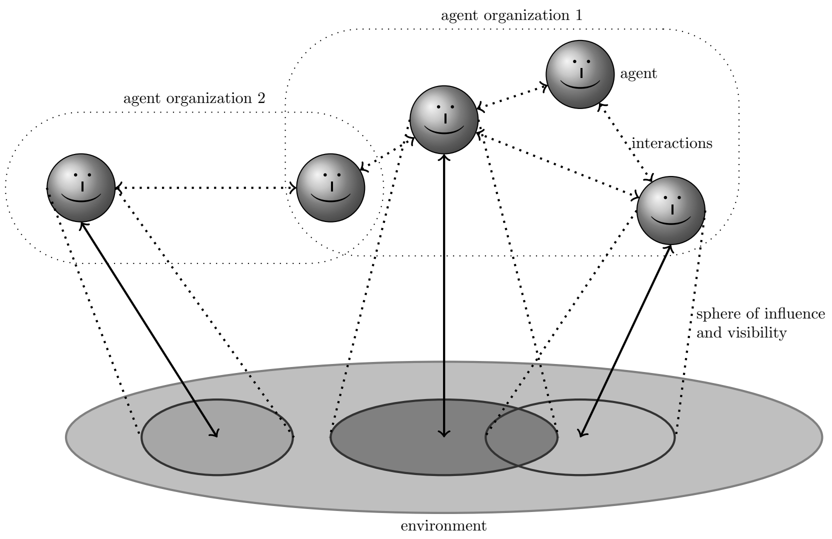


Figure 2.1: Characterisation of a multi-agent system. Adapted from (Jennings, 2000).

also implying asynchronous interactions as synchronizing them would result in drastic loss of performance.

Decentralized control. Control decisions must be taken locally in a distributed manner. A centralized controller would become a major bottleneck which, if it fails, might cause a failure of the whole system. Moreover, it is hard to see how one single agent could have complete, accurate information on the state of the entire system having a decentralized nature.

Inconsistent information. Because of the system's asynchronous and distributed nature the information that flows through the system coming from different sources may be inconsistent.

Arm-length's relationships. To promote information hiding and reduce the dependency between components information sharing proceeds by explicit communication between the components. That is to say, the components are at an arms-length relationship.

Continuous operation. Open systems often have a long uptime. The tasks of failing components are ideally taken over by other components such that failure of individual components does not compromise the continuation of the system's operation.

An additional, important requirement of open multi-agent systems that is not explicitly defined by Hewitt is described by Artikis and Pitt (2001):

Neutrality. An open multi-agent system should be neutral with respect to the internal architecture of their members. That is to say, agents are treated as black-boxes and no assumptions can be made about their internals.

Note how the characteristics of Jenning's and Wooldridge's (weak) notion of agency are in accordance with some of the above-mentioned characteristics of open systems. Autonomy, for example, accords with the notion of decentralized control dictating that decisions should be taken locally, whereas sociality accords with the idea of arm-length's relationships promoting direct communication. Also note that above characterization does not mention anything about whether agents may dynamically enter or exit, or are statically defined at runtime. This property is explicitly mentioned by Davidsson (2001) who defines the notion of openness by distinguishing between four different categories of multi-agent systems (societies in his terminology). That is, multi-agent systems can be categorized:

Closed. The set of agents that will participate in a system is known a priori and it is impossible for external agents to join.

Open. Any external agent can join the system without restrictions by starting to interact with some of the system's members.

Semi-closed. External agents cannot enter the system directly, but instead have the possibility to initiate a dedicated interface agent belonging to the system which acts on behalf of the external agent. Conformity to the society's restrictions is hard-wired in the interface agents.

Semi-open. External agents may directly join the society by contacting an authority responsible for deciding whether the agent is entitled to join. Once a society member an agent can directly interact with other members provided that the constraints imposed by the system are respected.

Note that Davidsson's classification is orthogonal to Hewitt's and Artikis/Pitt's characterization. Indeed, the characteristics they attribute to open systems are not unique to open systems alone. A closed system, for example, can still be characterized by a continuous operation and distribution of the components in the system.

Besides the collection of interacting agents, the environment plays an important role in the development of multi-agent systems. In Wooldridge's definition of agent, for instance, it is explicitly mentioned that agents are situated in an environment. That the environment is more than merely an agent container becomes apparent from the definition of Weyns *et al.* who define an environment as:

> "a first-class abstraction that provides the surrounding conditions for agents to exist and that mediates both the interaction among agents and access to resources." (Weyns et al., 2007)

This definition stresses that the environment is an essential part of a multi-agent system. It is a first-class citizen encapsulating its own domain specific functionality and is developed independently from the agents that reside in the multi-agent system. Weyns *et al.*attribute the following responsibilities to environments in the context of multi-agent systems:

Structuring. The environment structures a multi-agent system by means of its physical structure (e.g. distribution), by structuring the communication messages that flow through the environment, and by structuring the social relationships between the agents in terms of groups and roles.

Resource and service embedding. An environment encapsulates resources and functionality that agents may exploit in pursuit of their objectives.

Maintaining dynamics. The environment may have its own dynamics separately from the agents' activities (e.g. a daemon that automatically performs a daily task).

Local observability. Agents may inspect parts of the environment depending on their sphere of visibility (see figure 2.1).

Local accessibility. Agents may alter parts of the environment determined by their sphere of influence (see figure 2.1).

Regulating. The environment may regulate the agents' behavior by means of rules. The sphere of influence could, for example, be determined by the rules stating which parts of the environment an agent may alter and perceive.

Centered around the same principle that an environment implements non-agent functionality agents may exploit in achieving their goals, is the view of Ricci *et al.*(2006; 2007) who apply the real-life metaphor of humans who use non-autonomous tools in achieving their objectives on multi-agent systems. Just like Weyns and colleagues they argue that a multi-agent system should not only be composed of agents, but should also include the notion of artifact as a first-class abstraction. An artifact is

> "a computational device populating agents' environment, designed to provide some kind of function or service, to be used by agents – either individually or collectively – to achieve their goals and to support their tasks." (Ricci et al., 2006)

An artifact comes with a usage interface defining the operations which agents can invoke and operating instructions describing how to use these operations to fruitfully exploit the artifact's functionality. What, according to Ricci *et al.* (2006), discerns an artifact from an agent is that an artifact is merely a tool that agents use by performing operations on it. Contrasting an agent an artifact is neither

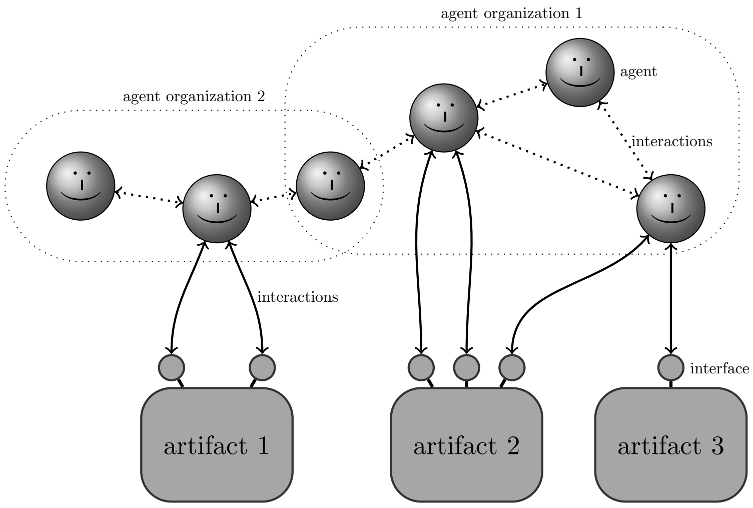


Figure 2.2: Characterisation of a multi-agent system adopting the agents and artifacts (A&A) approach.

meant to be autonomous, nor pro-active, nor social. The decision to use a particular artifact can be hard-wired in an agent's plan, but can also be the result of a dedicated process in which the agent at run-time discovers artifacts and decides which ones to use. Of course, the latter requires advanced reasoning capabilities from an agent. The approach in which a multi-agent system is enriched with artifacts is called the agent and artifacts (A&A for short) approach to engineering multi-agent systems (Ricci et al., 2007), and is depicted in figure 2.2.

2.3 Organization-Oriented Multi-Agent Systems

The agent-oriented paradigm is a potential powerful means to build systems in which autonomous, heterogeneous and independently designed agents may dynamically enter and exit. The view on a multi-agent system sketched by figure 2.1 supposes that how the agents are organized, i.e. who to interact with and how, is decided by the agents themselves. However, adopting a view in which all the agents may interact freely without any form of external control and without taking the possibility of self-interested behavior into account leads to the following problems:

- Because agents are autonomous and decide at runtime who to interact with, the patterns and the outcomes of their interactions are inherently unpredictable (Jennings, 2000);
- The behavior of the overall system based on its constituent components is

extremely difficult to predict as a consequence of the likelihood of emergent and possibly unwanted behavior (Jennings, 2000);

- The absence of external control compromises security as the agents could easily revert to malicious behavior (Ferber et al., 2003);
- Because agents are accessible from everywhere, it is difficult to apply modularization on the multi-agent system, e.g. grouping together agents that have to interact to achieve their goals (Ferber et al., 2003).

Adopting an organizational perspective, i.e. analyzing and designing a multi-agent system as a computational organization using an explicit notion of social concepts such as roles and norms, has been widely promoted as a potential candidate to overcome these difficulties (cf. (Jennings, 2000; Zambonelli et al., 2001b; Ferber et al., 2003; Dignum, 2004, 2009a)). Jennings (2000) observed that the organizational view is naturally reflected by the way in which agents interact and behave to achieve their objectives, because these interactions are often determined by their underlying relationships. A typical example of such an organizational relationship is a manager and subordinate relation.

In accordance with Weyn's notion of an interaction layer, many researchers have introduced approaches in which an organizational infrastructure is deployed to take charge of managing the interactions determined by the agents' relationships. Agents then dynamically enter and exit the system by enacting and de-enacting roles that are explicitly defined at the organizational infrastructure. They interact via the organizational infrastructure, a computational medium that regulates the agents' interactions based on explicitly defined protocols associated to the roles they play.

A conceptual view of this organizational approach is depicted by figure 2.3. The advantage of the organizational perspective is that it decreases the system's complexity and eases the management of multi-agent systems, because it leads to a cleaner separation between the agent-level and organization-level dimensions (Zambonelli et al., 2001b). Another advantage is that the organizational view accords more with how we as humans understand and structure the world around us. When the system is designed to support some real-world organization such as an online auction or a conference management system, the organizational stance reduces the conceptual distance between the software system and the real-world system it has to support (Zambonelli et al., 2001b).

Despite (or rather because of) its wide use, terminological and conceptual confusion exists when it comes to modeling agent organizations. The confusion starts with the concept of "agent organization". Sometimes this concept is used to mean the whole society consisting of interacting agents (and possibly also including non-agent entities), irrespective of whether organizational concepts were actually used in developing them and can be explicitly found. The term is also often used to mean a computational entity (or organizational middleware or organizational artifact) that is developed apart from the agents with the aim of

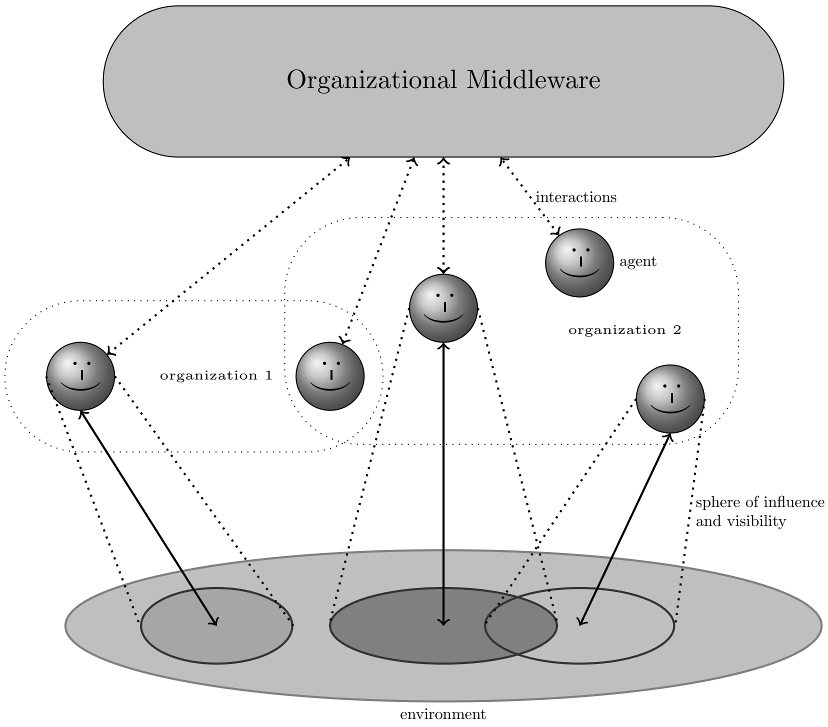


Figure 2.3: Characterisation of a multi-agent system in which groups of interacting agents are conceived as organizations. Adapted from (Zambonelli et al., 2001b).

regulating the agent's behavior by explicitly using organizational constructs such as roles and norms. In what follows we will use the term agent organization to denote the first and organizational middleware or artifact to denote the latter. Note that an agent organization may (or may not) include organizational middleware. In an attempt to clarify some of the confusion about what constitutes an agent organization, Coutinho *et al.* (2005) performed a comparative study of existing organizational models to formulate a preliminary ontology of modeling dimensions of agent organizations. They identify the following key dimensions of an organization:

Structural dimension, e.g. groups, roles and their relations;

Normative dimension, e.g. norms, obligations, prohibitions;

Functional dimension, e.g. global goals, shared plans;

Dialogical dimension, e.g. communicative acts, communication protocols.

In this thesis we mainly focus on the structural and normative dimension of an agent organization. At these dimensions we find the concepts that arguably can be considered the cornerstones of every agent organization, namely *roles* that agents can play and *norms* that describe how the agents should behave. Both concepts will be explained in more detail below. Orthogonal to these dimensions we find the "dynamics" pertaining to how the elements at each of these dimensions may evolve during the organization's life-time. As we will see in a moment, the dynamics are to a large extent explained by the structure and semantics of the organizational constructs.

The functional dimension provides functionalities an agent may exploit in achieving her objectives which she could not achieve on her own. At this dimension we often find concepts such as a task (or goal) decomposition tree. Such a decomposition tree typically breaks a more abstract goal into a sequence of more concrete goals (or even plans and single actions) that need to be achieved (or performed) in achieving the more abstract top-level goal. A task (or goal) decomposition tree can be considered a global plan describing how a (common) goal can be achieved by a group of agents by executing the plan together. Examples of such approaches are the STEAM framework of Tambe *et al.* (1998) and the functional specification of the MOISE+ model by Hübner *et al.* (2002). These approaches usually assume the agents to be cooperative, an assumption that cannot be made for open systems in which little is known about which agents will participate and about the behavior they will exhibit. Therefore, this concept will not be the main focus of this thesis. That does not mean, however, no means of providing functionality to the agents is provided in our approach. Why else would the agents even bother obeying the norms? Functionality can be provided in many ways, e.g. the artifacts are tools that can be used by the agents and, as we will show later on, our view on roles (presented in chapter 5) provides the agents functionality they may exploit.

2.4 Norms

Norms are a recurring concept in a broad set of different fields ranging from sociology to computer science. Roughly, two types of norms can be distinguished, viz. social norms and legal norms. Britannica online encyclopedia defines the concept of social norm as a:

> "rule or standard of behavior shared by members of a social group. Norms may be internalized, i.e., incorporated within the individual so that there is conformity without external rewards or punishments, or they may be enforced by positive or negative sanctions from without. The social unit sharing particular norms may be small (e.g., a clique of friends) or may include all adult members of a society. Norms are more specific than values or ideals: honesty is a general value, but the rules defining what is honest behaviour in a particular situation are

> norms.
> ...
> Norm is also used to mean a statistically determined standard or the average behaviour, attitude, or opinion of a social group. In this sense it means actual, rather than expected, behavior."

Social norms are thus the rules that describe what is considered appropriate (or inappropriate) behavior shared by members of a society or a group. Typical examples of social norms are to wear a ceremonial dress at a grand ball, and to take a taxi home if you are not sober. Closely related is the concept of legal norm or law, which is defined by Britannica online encyclopedia as:

> "the discipline and profession concerned with the customs, practices, and rules of conduct of a community that are recognized as binding by the community. Enforcement of the body of rules is through a controlling authority."

The difference between social and legal norms thus seems to be that with a legal norm an enforcing authority is involved. Yet, the distinction between social and legal norms is not that clear-cut. In fact, some social norms may become legal norms when they are established as laws that are enforced by some legal authority. For example, in many countries it is forbidden by law to drink and drive. Note that not all social norms are legal norms, e.g. to dress appropriately at a grand ball is usually not prescribed by law. Also the opposite, i.e. all legal norms emanate from social norms, does not generally hold, e.g. in some societies it is accepted (albeit forbidden by law) to drink and drive. Closely related to the distinction between social and legal norms are the different views that can be adopted, namely the legalistic view and interactionist view (see also (Boella et al., 2008)). In fact, the legal norms accord with a legalistic view, whereas social norms accord with a interactionist view. In the legastic view a top-down view on the norms is adopted, presupposing a controlling authority regulating the agents' behavior by motivating them to abide by the norms by rewarding their compliance and punishing their wrongdoings. In the interactionist view, a bottom-up perspective is taken, in which no regulating authority is assumed. Instead, norms may emerge from the interactions and behavior shared by a group of agents. These norms are abided by because they contribute to (or at least not obstruct) the goals of the individual agents or because the agents share the same values. In this thesis we are interested in exogenous coordination artifacts that use norms to govern the agents' behavior. It should thus be noted that we are not after mimicking social or legal laws; we are after developing a programming language that is merely inspired by these concepts. Bearing that in mind, it is interesting to note that because an organizational artifact can be considered an authority that enforces norms, our view can be best compared with the legalistic view on norms. The reasons for adopting this view will be motivated later on.

The above discussion of norms reveals a key aspect that has also been recognized by the field of philosophy. That is, norms do not describe how the world *is*, but rather prescribe how it *should be.* This implies that there can be a discrepancy between the actual behavior and the desired behavior, i.e. norms can be violated. Originating from philosophy norms have found their way to the field of computer science and multi-agent systems research through the field of deontic logic, a research field that endeavors to capture normative concepts such as obligation, prohibition and permission formally in modal logic (Chellas, 1980). Indeed, Jones and Sergot (1993) have illustrated that a computer system can be seen as an instance of a normative system, that is, a system in which actuality and ideality do not necessarily coincide. Based on this convergence between actuality and ideality Meyer and Wieringa (1993, preface) explain the importance of using normative concepts in the specification of computer systems and the role of deontic logic in doing so:

> "Until recently in specifications of systems in computational environments the distinction between normative behavior (as it *should be*) and actual behavior (as it *is*) has been disregarded: mostly it is not possible to specify that some system behavior is non-normative (illegal) but nevertheless possible. Often illegal behavior is just ruled out by specification, although it is very important to be able to specify what should happen if such illegal but possible behavior occurs! Deontic logic provides a means to do just this by using special modal operators that indicate the status of behavior: that is whether it is legal (normative) or not." (Meyer and Wieringa, 1993, preface).

In an open multi-agent system in which little assumptions can be made about the participating individuals, non-normative behavior is likely to occur. From the fact that no full control can be exerted over the individual agents (as is the case in a system that is characterized by some degree of openness) it follows that non-normative behavior cannot always be prevented from happening. Consequently, in engineering such systems it becomes even more important to distinguish between actual versus ideal behavior. Indeed, it has been observed in the AgentLink Roadmap (Luck et al., 2003, Fig. 7.1.) that norms must be introduced in agent technology and the use of norms for analyzing and designing software systems has gained attention from many multi-agent system researchers.

The interest in the use of norms in multi-agent systems has led to the emergence of the research area of normative multi-agent systems, abbreviated as NorMAS. The area of normative multi-agent systems can be seen as the intersection of the research areas of normative systems and multi-agent systems. A definition of a normative multi-agent system that has a broad support in the NorMAS community can be found in (Boella et al., 2006), in which a normative multi-agent system is defined as:

> "A normative multi-agent system is a multi-agent system together with normative systems in which agents on the one hand can decide whether to follow the explicitly represented norms, and on the other

the normative systems specify how and in which extent the agents can modify the norms." (Boella et al., 2006)

Three important observations can be made about this definition. A first observation is that norms are assumed to be *explicitly* represented in the system. One reason for representing norms explicitly is given by Boella *et al.* (2009) who argue that *"[a]ny requirement can be seen as a norm the system has to comply with [..., but] [c]alling every requirement a norm clearly make the concept empty and useless."* Moreover, an explicit representation opens up the opportunity for agents to observe and reason about the norms, although approaches to agent programming that aim at the internalization of norms and the agent's ability to reason with them are still scarce. An example of work in this direction is the work of Meneguzzi *et al.* (2009). From a software engineering perspective, having norms as first-class citizens eases understandability and manageability of the system. A second observation is, that in accordance with Meyer and Wieringa's observation, agents can (at runtime) decide whether to follow the norms or not, and consequently norms can be violated. Clearly, considering external agents over whom little control can be exerted, the system cannot always coerce preferred behavior. A normative system can, for example, only oblige an agent to perform an action, but cannot force it to do so. Even in cases where a system can do so, having the possibility to violate norms increases the choices agents have and thereby their autonomy, as for example argued in (Aldewereld, 2007; Castelfranchi, 2004). A third and final observation is that at runtime norms can be changed by the agents or the system. Facing the unpredictable and dynamic nature of the environments normative multi-agent systems are typically deployed in a static view in which the norms are specified at design time and cannot be modified at runtime does often not suffice (Castelfranchi, 2000; Boella et al., 2006; Dignum, 2009b). Modifying norms at runtime increases the system's flexibility to manage a priori unforeseen situations. Although a more elaborate discussion on how norms may be changed at run-time is postponed to chapter 6, we do note here that an explicit representation of the norms is necessary for them to be modified at runtime.

In what follows we give a concise overview of the key issues involved in using normative concepts for the analysis, design – and most importantly – implementation of computational multi-agent systems. Because of its intimate relation with norms as used in agent organizations, we start with a brief discussion on some of the relevant logical formalisms which try to capture normative concepts.

2.4.1 On the Logic of Norms

Because it is generally hard to thoroughly explain every-day life concepts in natural language, researchers have tried to capture them in a formal system allowing for a rigorous study of their key features. Deontic logic is the field of logic that formally studies the concepts of obligation, prohibition and permission. What is considered the first real system of deontic logic was proposed by Von Wright

(1951) which evolved into what is now known as Standard Deontic Logic (SDL for short). Introducing the modalities O for obligation, P for permission and F for forbiddance, SDL is a system that consists of the following axioms and rules:

(Taut)	All (or enough) tautologies of propositional calculus
(K_O)	$O(p \rightarrow q) \rightarrow (Op \rightarrow Oq)$
(D_O)	$\neg O\bot$
(P)	$Pp \leftrightarrow \neg O \neg p$
(F)	$Fp \leftrightarrow \neg Pp$
(MP)	$p, p \rightarrow q \vdash q$ (Modus Ponens)
(N_O)	$p \vdash Op$ (O-Necessitation)

From these axioms and derivation rules theorems can be derived, amongst which:

(1) $Op \leftrightarrow \neg P \neg p$
(2) $O(p \wedge q) \leftrightarrow Op \wedge Oq$
(3) $O\neg p \rightarrow O(p \rightarrow q)$
(4) $Op \rightarrow O(p \vee q)$

Most of these theorems seem reasonable, such as (1) which tells us that you cannot be obliged to do p and at the same time not permitted to do so.[1] They also include, however, some rather paradoxical ones. Most of these paradoxes have to do with how the derived formulae are read. If, for example, $O(p \rightarrow q)$ is (wrongly) read as an obligation to do q if we do p, then (3) tells us that violating an obligation by not refraining from p commits us to any (absurd) action, e.g. to fly over the moon. The last one (4) is known as Ross' paradox, which can be illustrated as "if one is obliged to mail a letter, one is obliged to mail the letter or burn it", which sounds strange indeed. Much of deontic logic has been concerned with offering alternative logics with the goal of solving the paradoxes and increasing expressiveness. What follows is a brief, non-exhaustive discussion of some of these attempts that are relevant to the field of multi-agent systems and, in particular, to this thesis.

Another important problem in deontic logic is that of how to represent conditional obligations. That is to say, obligations that only hold under certain conditions. For example, an obligation expressing that "if you are assigned a paper to review, then you ought to read it." From (semi-)paradoxical theorem (3) we know that expressing it as $O(assigned \rightarrow read)$ can be problematic. It might seem adequate to express this sentence in SDL as $assigned \rightarrow O(read)$, but also this is not satisfactory. Under this representation we are vulnerable to what is known as Forrester's paradox of gentle murder. This paradox is illustrated as "it is forbidden to murder, but if you murder, you ought to do it gently." The sentences before are expressed in SDL as $F(murder)$ and $murder \rightarrow O(murdergently)$. Suppose we adopt the macabre assumption that we indeed turn homicidal, i.e. adopting a fact $murder$, and adopt the plausible assumption that gentle murder implies mur-

[1]Another possible reading of these statements is "to have p" interpreting p as a description of a state rather than the performance of an action as is the case in the reading "to do p". This pertains to the distinction between ought-to-do and ought-to-be statements, cf. (d'Altan et al., 1996).

der, i.e. $murdergently \rightarrow murder$. Even though all the aforementioned sentences make perfect sense, put together in SDL they are inconsistent. The gentle murder paradox illustrates SDL's disability to express contrary-to-duty obligations, that is, obligations that only arise in the sub-ideal situation some other norm is violated.

Another attempt to formally capture norms is the one of Anderson who attempted to reduce deontic logic to the standard modalities of necessity (expressed by $\Box$) and possibility (expressed by $\Diamond$) by introducing a special propositional atom V denoting a violation (or the liability to sanction). In this system, an obligation Op is reduced to $\Box(\neg p \rightarrow V)$ meaning that not p necessarily implies a violation. A prohibition Fp is reduced to $\Box(p \rightarrow V)$ meaning that p necessarily implies a violation. It turned out, however, that also this system is vulnerable to some nasty paradoxes (cf. (McArthur, 1981)).

Enriching deontic logic with conditional obligations and prohibitions can without doubt be called a troublesome exercise. Another notoriously difficult problem is that of how to express deadlines. In SDL having an obligation to do p means having an obligation to do p *now*. However, an obligation is usually associated with a deadline that specifies when an obligation should have been fulfilled. Examples of research that focuses on the expression of obligations with deadlines are the work of Dignum *et al.* (2004) and Boella *et al.* (2008). Boella and colleagues, for example, introduce a logic of conditional obligations with deadlines. The obligations take on the form of triples (c, x, d) with the intuitive reading that "under condition c an obligation can be entailed to establish situation x before deadline d holds."

Much of deontic logic has been concerned with the modalities of permission, obligation and prohibition that specify what is considered desired and undesired behavior. Norms that include these modalities serve to regulate the behavior of a society, and consequently, are typically being referred to as *regulative* norms. It has been recognized that in specifying normative systems, besides regulative norms, also constitutive norms are needed that define that something *counts as* something else in a given society (cf. (Searle, 1995; Jones and Sergot, 1996; Boella and van der Torre, 2004; Grossi, 2007)). To understand constitutive norms, it is important to understand Searle's (1995) distinction between *brute* and *institutional* facts. Brute facts describe the observable part of the world, whereas institutional facts describe things that only exist in a society with certain conventions, rules, or norms in place. Holding a banknote is an example of a brute fact (it is observable); that this banknote is money that is generally accepted as payment for goods and services is an institutional fact (it is not observable and does only hold for particular societies). Constitutive norms then establish the institutional facts by describing what (described by brute and constitutive facts) counts as what (described by institutional facts) in a given society. For example, that giving someone money counts as a payment. Constitutive norms are typically expressed as so-called counts-as rules that take on the form "X counts as Y in context C."

It is important to note that regulative and constitutive norms are intimately related. Suppose, for example, we defined a regulative norm expressing an obligation to have made a payment. We can then use constitutive norms to specify that (say) both transferring money by bank and paying in cash counts as a payment. The intimate relation between constitutive and regulative norms also comes to light in the work of Grossi (2007) who investigates different facets of counts-as rules from a logical perspective, and even shows how counts-as rules can be used to express regulative norms by using the same notion of violation as coined by Anderson. Intuitively, the deontic notion of being prohibited to be in a state characterized by X can be modeled in terms of counts-as statements by stating that X counts-as a violation. If it is permitted to be in a situation in which X holds then being in a situation in which X holds does not necessarily count as a violation.[2] An obligation to be in a state in which X holds means that being in a state in which X does *not* hold necessarily counts as a violation.

All the debate about which primary concepts to include in a logic of norms and the unsolved problems that have plagued deontic logic for decades now might raise the question if norms and deontic logic can be used in practice, e.g. for the implementation of computational organization-oriented multi-agent systems? Meyer et al. (1994) give an answer to this question by arguing that:

> "although [...] not all long-standing problems become clear or irrelevant when looking at these matters from the perspective of computer science, it is nevertheless undoubtedly the case that some of the problems (i.e. paradoxes) lose their bite when one has the less ambitious motives to use deontic logic and one is interested in such prosaic things like a robot system in which a robot is allowed/forbidden to do certain actions rather than the use of deontic logic for reasoning about moral behavior of man."

It seems that none of the above logical formalisms can be used directly when it comes to using them to implement the normative component of computational agent organizations. We are, for example, not in the first place interested in things such as expressing nested obligations (e.g. OOp) and making logical inferences based on a set of deontic expressions. Moreover, using one of the formalisms to implement the normative component of an agent organization demands advanced mathematical skills of the programmer. For a possible success of norms in an organization-oriented programming language, we strongly feel that simplicity is key! For our purposes we do not need full-blown deontic logic. On the other hand, as we shall see below, implementing norms in computational multi-agent systems raises new issues that were originally not considered by the field of deontic logic.

[2]In (Grossi, 2007) also a more strong notion of obligation is defined in which p is permitted iff p necessarily counts as $-viol$.

2.4.2 Towards Norms in Computational Multi-Agent Systems

The use of norms in the implementation of multi-agent systems can be traced back at least as far as (Shoham and Tennenholtz, 1995) by Shoham and Tennenholtz who suggested an approach in which the agents of the system are implemented to respect a set of social laws that are adopted by their designers. This means that, in their view, the whole multi-agent system is developed off-line and laws are devised the agents should follow in order to solve the coordination problem at hand. The agents are then programmed such that they only exhibit norm-abiding behavior. In other words, the protocols are hardwired in the agents and no explicit representation of the norms is needed.

An approach in which the whole system is designed off-line accords with the view of a closed system (Davidsson, 2001). Clearly, the assumption that all the agents are designed to obey the norms cannot be generally made for open multi-agent systems in which agents that are designed by developers different from that of the overall system and/or other agents dynamically enter and leave. These agents might be endowed with objectives that jeopardize those of the global system. Moreover, as Dignum (1999) argued, hardwiring the social rules into the agents compromises their autonomy and their flexibility to respond to environmental changes and other agents violating the rules. The agent's autonomy is limited because they cannot deviate from the expected behavior expressed by the norms and their flexibility is compromised because they react in a predictable manner dictated by the protocol. To overcome these issues Dignum proposes the *explicit* use of normative concepts such as obligations, prohibitions and permissions to govern the agents' behavior. These norms should then be defined as a separate component external to the agents that are assumed to have the ability to internalize the norms in order to decide which ones to follow and which actions to take. The issue of agents taking norms into account in their reasoning and decision-making process is related to the implementation of norms from the perspective of an agent (examples of work on this perspective are (Broersen et al., 2002; Meneguzzi and Luck, 2009; Lopez y Lopez et al., 2006)). To separate the norms from the agents is related to the implementation of norms from the perspective of the organizational artifact, i.e. the norm-enforcing authority. We emphasize that both the agent as well as the organizational artifact's perspectives are important for a successful use of norms in the implementation of multi-agent systems. However, in this dissertation we focus on the implementation of norms from the organization's perspective in which there is a computational entity (an organizational artifact) that acts as an enforcing authority.

When taking an organizational perspective, there remain some questions to be answered. For example, "who is responsible for creating the norms and how can they be enforced?". Below we discuss these questions.

Norm legislation: off-line versus online design

When adopting an organizational perspective on the norms, there is still the question of who legislates the norms, i.e. who is in charge of making the rules that are to be enforced by the regulating authority? There are roughly two views on this issue. In the first view is closely related to the interactionist view [3] (see (Boella et al., 2008)) in which it is assumed that the agents themselves are in charge of creating the norms. A second view, the one we will adopt in this thesis, is the view in which the norms are designed off-line by the system's designer. Even though it can be argued that this reduces the system's flexibility and is not the only mechanism that is needed to guarantee systems to work as desired (cf. (Castelfranchi, 2000)), taking a design stance in creating the norms has numerous advantages as, for example, claimed by Moses and Tennenholtz, and Artikis and Sergot:

- At design time, the designers of the system usually have a greater number of resources at their disposal (think of computational power, information and time), than the agents have at run- time. Therefore, some problems may be better solved by the designers at design-time than they would be solved at run-time by the agents. (Moses and Tennenholtz, 1995)

- When effective norms are hard to specify, the environment may be exploited and modified (at design-time) such that the task of devising social constraints is simplified (e.g., the addition of traffic lights to an environment in which agents move). (Moses and Tennenholtz, 1995)

- If the rules the agents should adhere to are a priori known, many conflicts might be avoided when the agents (at design-time or run-time) produce their strategies such that the rules are respected. This could significantly reduce the number of run-time conflicts. (Moses and Tennenholtz, 1995)

- Publishing the social constraints facilitates agents in the process of deciding whether to start their interactions in a system or not. Likewise, decisions can be made by the designer of the agent whether to deploy an agent in a system at all. (Artikis and Sergot, 2010)

In addition to these observations, we argue that having a system in which the norms spontaneously emerge diminishes the predictability of the behavior of the system a whole. As already discussed above, according to Jennings (2000) this is exactly one of the major problems in the design of multi-agent systems we are trying to overcome by deploying norms in the first place!

[3] It should be noted that in the interactionist view there is often no enforcing authority involved.

On the representation of enforceable norms

Irrespective of whether we let the norms be created by the agents or the system's designer, the issue of how they should be represented for their implementation in computational multi-agent systems is still open. We already argued that borrowing a logical formalism of norms does not suffice when it comes to operationalizing norms for their use in computational multi-agent systems. Instead, different representations than typically used in, for example, deontic logic are needed together with techniques for enforcing them. An extensive analysis of the representation of norms for their use in multi-agent systems was done by Lopez y Lopez and Luck (2006). Their analysis covers a wide range of topics involving in the area of normative multi-agent systems. Here, however, we will concentrate on their definition of norm. In their view, a norm comprises the following components:

Normative goals describing the things that ought to be done, i.e. states to be achieved (obligations) or avoided (prohibitions).

Addressees relating to the agents responsible for satisfying the normative goals.

Beneficiaries relating to agents that benefit from the satisfaction of the normative goals.

Context specifying when the norm is active, i.e. in which context the normative goals should be satisfied.

Exceptions describing the situations in which addressees are not liable to punishment in case of non-compliance.

Rewards specifying the incentives for complying with the norm.

Punishments specifying the penalties that are imposed in case of violation.

In addition to these components, Lopez y Lopez and Luck, define the notion of interlocking of norms. That is, norms that are activated when some other norm has been complied with or has been violated. A norm which (non-)compliance gives rise to another norm is called the primary norm, whereas the norm that is activated as a result of either the fulfillment or violation of the primary norm is called a secondary norm. Secondary norms are linked to primary norms by means of their context. Note that contrary-to-duty obligations (Prakken and Sergot, 1996) are a special case of norm interlocking. A contrary-to-duty obligation is an obligation one obtains after violating some other norm. Interlocking pertains to the more general case also including, for example, an obligation (or prohibition) one obtains after the fulfillment of some norm.

Enforcing the norms

The analysis in (Lopez y Lopez et al., 2006) is merely syntactical. That is to say, no information is provided on how to operationalize the norms for their use in a multi-agent system. In contrast to (Lopez y Lopez et al., 2006), Vázquez-Salceda *et al.* (2008) propose a syntactical format of norms (one that roughly includes the same components as that of Lopez y Lopez and Luck (2006)), together with some guidelines on how to implement norms in multi-agent systems. They observe that the operationalization of norms requires an implementation of a mechanism of norm enforcement as a process that is responsible for detecting when a norm is active, detecting violations of the norms and handling of these violations. How such a concrete mechanism is implemented depends on the level of control that can be exerted over the agents and the verifiability of the contents of the norms (i.e. the conditions and actions the norm refers to) (Vázquez-Salceda et al., 2004). In (Vázquez-Salceda et al., 2004) different types of norms are distinguished relating to the level of control that can be exerted over the norm's addressee, namely norms concerning:

External agents that are developed apart from the organization enforcing the norms. A further distinction is made between norms concerning behavior that is directly observable by the organization (e.g. a norm stating that a review should be uploaded to the system) and norms concerning behavior that is not directly observable by the organization (e.g. a norm stating that proceedings should be sent to all authors of accepted papers);

Internal agents that are developed by the same party as the organization, including agents responsible for enforcing the norms.

In case of internal agents, one might resort to a norm-obedient implementation of these agents. (Although it should be noted that the above explained disadvantages still apply.) In this thesis, however, we will also be concerned with external agents, explaining the need for a norm enforcement mechanism. Regarding the levels of verifiability of the contents of a norm, Vázquez-Salceda et al. discern three different types, viz.:

Computationally verifiable , meaning that the condition or action the norm refers to can be directly verified by the organization. The before-mentioned norm concerning the review that should be uploaded to a database is an example of a norm that is computationally verifiable.

Non-computationally verifiable , i.e. the condition or action cannot be (easily) verified by the organization, but extra resources are needed to check for compliance. An example is a norm phrasing that reviewers should found their claims with facts.

Non-verifiable , meaning that the condition or action cannot be verified by the organization. An example of a non-verifiable norm is one expressing that an agent is not permitted to have certain beliefs or goals in case its mental state cannot be inspected (a plausible assumption in open systems!).

When the norms are about actions or conditions that are not directly observable by the organizational artifact, it should rely on information from trusted parties to check for compliance. To verify, for example, a norm expressing that a hard copy of this thesis should be sent to each member of the reading committee, the system could rely on information about the delivery status provided by the courier service. This stresses the need for a coordination artifact to have the possibility to be informed about facts about the outside world by (trusted) external agents. When norms are computationally verifiable, there are basically two ways of motivating agents to abide by the norms, viz. by using *regimentation* and *enforcement* of the norms, see also (Aldewereld, 2007; Castelfranchi, 2000; Grossi, 2007; Jones and Sergot, 1993).

Regimentation, as coined by Jones and Sergot (1993), is based on the idea of *ruling out* all the actions that will lead to an undesired state, such that a violation of the norms will never happen. An illustrative description of the concept of regimentation is given by Castelfranchi:

> "By establishing certain 'material' conditions in the environment that the agent is not able to change, I prevent it from doing a given action (or even from attempting to do it), and at the same time -since it is motivated to achieve a given goal- I channel its behavior in a given direction (towards the remaining practicable solutions) i.e.; towards what is practically permitted. Sometimes towards the only remaining possible move, that becomes 'obligatory' since there are no degrees of freedom for the agent. For example, if you put me in a tube, and I want to go out, and you block one access (my way in), I'm obliged or better constrained to go straight ahead, as you want." (Castelfranchi, 2000)

Regimentation thus implies that the system can somehow prevent an agent from performing an action that causes a violation. An example of a case in which this presumption holds is an operating system that can disable certain operations for users that do not have the right permissions. Note that this presumption does not imply that the system has control over the internals of the agent, it can still try to perform the operation, but the result is simply not effectuated by the operating system.

As alternative to regimentation, enforcement can be used. Enforcement is based on the idea of responding after a violation of the norms is detected. Following the old Roman saying "ubi lex ibi poena" (where there is a law, there is a sanction), agents are motivated to abide by the norms by rewarding good behavior and punishing bad behavior. Remember that even when regimentation can be applied, enforcement might be used, because having the possibility to violate norms increases the choices agents have and thereby their autonomy, as for example argued by Aldewereld (2007) and Castelfranchi (2004).

2.4.3 Normative Frameworks in Multi-Agent Systems

In recent years a vast amount of computational frameworks have appeared in literature that use normative concepts to regulate agents' behavior. As already mentioned in the introduction, also outside the area of multi-agent systems normative concepts have appeared, e.g. in the area of usage control (Park and Sandhu, 2002) the use of obligations has been mentioned (without explaining how to enforce them) for describing the requirements an individual has to fulfill after having gained access to a resource. In this section we discuss (in no particular order) some examples of the approaches from the area of multi-agent systems that we believe are most related to our work. Considering the large amount of research in this direction, an exhaustive survey seems impossible. The discussion that follows is merely intended to reveal the key issues that motivate our work in the direction of normative multi-agent systems. It is fair to say that many of the works mentioned here also include other organization-oriented concepts that will not be considered in this thesis, e.g. the functional and dialogical dimension as identified by Coutinho *et al.* (2005). Here, we will primarily focus only on the normative dimension of these frameworks and only mention the other aspects if strictly needed. In the next section we will elaborate upon the concept of role that is a key ingredient of the structural dimension.

Electronic Institutions

A strand of work that primarily focuses on the normative dimension is the research on electronic institutions, computational interaction environments including a norm enforcement mechanism. One of the first platforms for executing electronic institutions is the AMELI platform (Esteva et al., 2004) of Esteva *et al.*, a middleware that regulates the agents' interactions. These institutions are specified in the graphical tool ISLANDER (Esteva et al., 2002) by the formal language presented in (Esteva et al., 2001). ISLANDER/AMELI views an electronic institution as a dialogic system in which the only interactions that take place in the system are speech acts (communicative actions) performed by agents. The interactions amongst agents take place in so-called scenes, group meetings agents can enter and leave and in which messages are exchanged. Associated to these scenes are communication protocols specifying the permitted dialogs by the agents interacting in the scene. More precisely, a communication protocol is a directed graph in which nodes represent the state of the interaction and its labeled edges define the speech acts that must be uttered to reach a next state in the protocol. How agents can legally move from scene to scene is specified by the performative structure, a network capturing the transitions agents can make between different scenes. Finally, certain circumstances in one scene (e.g., winning an auction) might lead to obligations in other scenes (e.g., an obligation to pay). Such obligations that are outside the scope of scenes are expressed as global norms that hold in the entire institution. These norms specify which obligations hold when certain speech acts have (not) been uttered in particular scenes. Just like the norms expressed by the protocols and the performative structure, these obligations can only refer to speech acts that should be uttered. A key aspect of the ISLANDER/AMELI approach is that it rules out all actions that do not conform to the specification. Agents cannot deviate from the behavior specified by the protocols and performative structure; only the paths of the directed graphs can be followed. Also the norms can never be violated; agents must typically have fulfilled all their obligations before they can proceed to other scenes.

Since its introduction multiple research has focused on extending the ISLANDER/AMELI framework with the purpose of increasing its expressiveness and flexibility. Garcia-Camino *et al.* (2005), for example, enrich the norms of ISLANDER/AMELI to include an explicit representation of obligations, prohibitions and permissions with conditional and temporal restrictions (deadlines). Moreover, they relax the assumption that norms can never be violated and introduce sanctions to punish agents that do not abide by the norms. This implies an enforcement rather than a regimentation strategy of the norms. The meaning of the syntactical constructs introduced in (Garcia-Camino et al., 2005) is explained by means of an implementation in the rule-based programming language Jess.

In both aforementioned approaches to electronic institutions the norms can only express which communicative acts an agent is obliged (permitted or forbidden) to perform. To overcome this limitation, the extension presented in (Garcia-Camino et al., 2005) has been further extended by Silva (2008) to also include norms that refer to non-dialogical actions. That is to say, actions that do not relate to the interaction between agents, but are performed in and have some effect on an environment the agents act upon. This allows for the expression of normative statements that describe a situation that should be established (or avoided) without referring to the actions that were used in reaching this situation. An example of such a normative statement is: "the total size of all uploaded files should not exceed 1 Gigabyte." There are many possible sequences of actions to reach a situation in which we exceed this limit, e.g. uploading one file of more than 1 Gigabyte, uploading two files of 600 MegaBytes, etcetera. With only actions to refer to such normative statements are hard (or even impossible) to express!

Norm-Governed Computational Societies in the Event Calculus and $\mathcal{C}+$

Another example of work that uses normative concepts to regulate the behavior of agents is the work of Artikis *et al.* who employ two formalisms for specifying and executing normative system specifications, viz. normative specifications in the event calculus (Artikis et al., 2009; Artikis and Sergot, 2010) and normative specifications in the action language $\mathcal{C}+$ (Artikis et al., 2009).

Event calculus is a logic programming formalism for representing and reasoning about actions (events) and their effects. Normative concepts including conditional obligations, conditional permissions and sanctions are directly specified in the event calculus. The programmer gives meaning to these normative concepts through a set of rules (axioms) that are programmed in the event calculus. Suppose, for example, we are to express that when a merchant M is offering a good G for price P at time T, then an accept action performed by costumer C will initiate an obligation to pay for the good. This obligation is revoked when the costumer performs the payment action. This can be specified in event calculus as follows:

$$\begin{gathered} \mathit{initiates}(\mathit{accept}(C,M,G,P),\mathit{obl}(C,\mathit{pay}(C,M,G,P)) = \mathit{true},T) \leftarrow \\ \mathit{holdsAt}(\mathit{offered}(M,G,P) = \mathit{true},T) \end{gathered} \tag{2.1}$$

$$\begin{gathered} \mathit{initiates}(\mathit{pay}(C,M,G,P'),\mathit{obl}(C,\mathit{pay}(C,M,G,P)) = \mathit{false},T) \leftarrow \\ \mathit{holdsAt}(\mathit{obl}(C,\mathit{pay}(C,M,G,P)),T), P' \geq P \end{gathered} \tag{2.2}$$

Violation of an obligation and the circumstances under which sanctions should be applied are defined in a similar fashion. A normative specification programmed in the event calculus is directly executable and the semantics of this program is thus determined by the semantics of the event calculus.

The implementation of a normative program in the $\mathcal{C}+$ action language proceeds in a similar fashion as implementing normative systems in the event calculus; the program is programmed directly in $\mathcal{C}+$ and the normative concepts obtain their meaning through the axioms as defined by the programmer. The $\mathcal{C}+$ specification can then be executed by the Causal Calculator (CCalc) software tool.

A particular strength of the CCalc are its prediction, planning and postdiction facilities. These enable (amongst others) the possibility to directly prove properties about a $\mathcal{C}+$ normative program in an automated way. This allows a designer to verify, at design-time, whether a normative program meets certain requirements, e.g. that a violation of an obligation will always be sanctioned. With an operational semantics this is not directly possible; to facilitate such automated verification tasks the operational semantics first needs to be implemented, see for example (Aştefănoaei et al., 2008; Dennis et al., 2009).

A key feature of the approach that is taken by Artikis *et al.* is that a normative program is directly implemented in the event calculus/$\mathcal{C}+$. The meaning of the normative concepts thus depends on the axioms that are programmed by the designer of the system. This gives a lot of freedom to the programmer in specifying the semantics of the normative concepts, but at the same time this implies that there is not one clear semantics attributed to the normative concepts the system designer can rely upon. Further, what can be implemented is restricted by what is expressible (syntactically and semantically) in these formalisms. It is, for example, not clear how (and if) the constructs for changing a set of norms at run-time (presented in chapter 6) can be implemented in the event calculus/$\mathcal{C}+$. In this thesis, we do not commit to a particular implementation formalism. We are interested in defining the syntax and operational semantics of a set of dedicated programming constructs by which normative programs can be implemented.

Another important observation to note (why this is important will be explained later on) is that similar to most work on electronic institutions (except for (Silva, 2008)), the normative concepts that are introduced in (Artikis et al., 2009; Artikis and Sergot, 2010) refer to *actions* an agent is obliged or permitted to perform. In (Sergot and Craven, 2006), however, Sergot and Craven extend the action language $\mathcal{C}+$ to include a native deontic component. In this extension it is also possible to express which *states* are obliged/permitted.

Commitment-Based Approaches

Another normative concept that shows a strong relationship with the notions of obligation and prohibition and that is often found in the area of computational multi-agent systems is that of commitment. Examples of this work on commitments are (Singh, 1999; Yolum and Singh, 2002; Fornara and Colombetti, 2009; Torroni et al., 2009). Even though (just like other normative concepts!) there seems to be no consensus on the exact meaning of a commitment, an often found interpretation is that they are obligations that are directed from one agent to another (Torroni et al., 2009). Commitments come in two forms: unconditional and conditional commitments. An unconditional commitment is of the form $C(x, y, p)$ meaning that the debtor x commits to creditor y to bring about the condition p. A conditional commitment is a four-tuple $CC(x, y, p, q)$ with the intuitive reading that if the condition p holds, then x will be committed to bring about the condition denoted by q.

Central to the approach of commitments is that agents manipulate them by means of special designated actions. More specifically, in (Singh, 1999) six actions are provided for manipulating commitments. Given that x, y, z are agents and c, c' are commitments, commitments can be manipulated through the following actions (taken from (Torroni et al., 2009)):

$create(x, c)$ establishes the commitment c. This operation can only be performed by the debtor of the commitment.

$discharge(x, c)$ resolves the commitment c. The discharge operation can only be performed by the debtor of the commitment to mean that the commitment has successfully been carried out. Discharging a commitment terminates that commitment.

$cancel(x, c)$ cancels the commitment c. The cancel operation is performed by the debtor of the commitment. Usually, the cancellation of a commitment is accompanied by the creation of another commitment to compensate for the cancellation.

$release(y, c)$ releases the debtor from the commitment c. It can be performed by the creditor to mean that the debtor is no longer obliged to carry out his commitment.

$assign(y, z, c)$ assigns a new agent as the creditor of the commitment. More specifically, the creditor of the commitment c may assign a new creditor z

to enable it to benefit from the commitment. Operationally, commitment c is eliminated and a new commitment c' is created for which z is the creditor.

$delegate(x, z, c)$ is performed by the debtor of commitment c to replace itself with another agent z so that z becomes responsible to carry out the commitment. Similar to the previous operation, commitment c is eliminated, and a new commitment c' is created in which z is the debtor.

Even though in (Singh, 1999) the relation between commitments and other normative concepts has been discussed, there still seems to be little (or no) consensus on their relation. Commitment-based approaches are particularly tailored towards modeling obligations between agents that arise from inter-agent communications and not so much towards expressing more abstract norms. It is, for example, difficult to see how a sentence saying that every paper that has been submitted to a conference should be reviewed by minimally three reviewers can be expressed by a commitment. It is these kind of normative statements we are interested in, in the context of this thesis. Moreover, commitment-based approaches assume agents to be capable of manipulating commitments directly, whereas in this thesis we will be primarily investigating the manipulation of obligations and prohibitions by an organizational framework, rather than by agents directly. A further discussion of the difference (or similarity) between the concept of commitment and other deontic concepts is beyond the scope of this thesis.

A Normative Programming Language based on Counts-As Rules

Inspired by the observation made by Grossi (2007) who has shown that counts-as rules can be used to express regulative norms also, in earlier work (Dastani et al., 2008, 2009) we presented a normative programming language based on counts-as rules. Inspired by Searle (1995) brute facts (first-order atomic formulae) describe the domain specific state of the environment agents act upon. Agents may manipulate the brute facts, e.g. uploading a paper in a database by means of performing actions. The normative program specifies how the brute state may evolve under the performance of actions. Counts-as rules are then used to normatively assess the brute facts and label a state with a normative judgment marking brute states as, for example, good or bad. For example a counts-as rule:

$$paper(X, Pages) \wedge Pages \geq 15 => viol(X, Pages, pagelimit) \tag{2.3}$$

states that uploaded papers that exceed 15 pages are marked as a violation of the page limit. The normative judgments about the brute state are stored as *institutional facts*, again taken from Searle (1995). To motivate the agents to abide by the norms, certain normative judgments might lead to sanctions which are imposed on the brute state. For example, rejecting a paper that violates the page limit by removing it from the database, which is expressed by a rule:

$$viol(X, Pages, pagelimit) => \text{not } paper(X, Pages) \tag{2.4}$$

Although this language is not as expressive as the normative languages we have discussed above in the sense that is lacks an explicit representation of the concepts of obligation, prohibition and deadline, it has two notable features. Firstly, similar to many agent-oriented programming languages, the syntactical constructs of the normative programming language are underpinned by an operational semantics (Plotkin, 1981). This has the advantage that the meaning of the language can be thoroughly described in a mathematical way that does not suffer from the ambiguity of natural language. Another benefit of an operational semantics is that it opens the way for model checking and formal verification. See for example (Dastani et al., 2008; Dennis et al., 2009) for work in this direction. Secondly, similar to the work of (Silva, 2008) our counts-as based language refers to states that agents can achieve in an environment they act upon, rather than actions. The rationale thereof has been explained above, and will be further explained in chapter 4 in which we extend the language to include an explicit representation of conditional obligations and deadlines. The solution provided in that chapter, however, does not share the capability of (Dastani et al., 2008, 2009) to express constitutive norms also. Another extension of the counts-as language can be found in (Tinnemeier et al., 2009a) in which we proposed the use of temporal operators (Emerson, 1990) in the antecedent of the counts-as rules. However, despite its expressiveness it turned out that this language demands advanced mathematical skills from the programmer. The normative language presented in chapter 4 is more practical in its use.

The Deontic Aspect of MOISE+

S-MOISE+ (Hübner et al., 2006b) is an organizational middleware following the MOISE+ model (Hübner et al., 2002). Besides a structural aspect defining the structure of an organization in terms of roles and inter-role relationships (more about that later on) and a functional aspect, the MOISE+ model includes a deontic aspect pertaining to the normative dimension of an organization. Because of the intimate relationship of the deontic dimension with the functional dimension, we first need a better understanding of MOISE+'s functional dimension.

MOISE+'s functional aspect is concerned with the functioning of the organization. It provides the agents with information about how more abstract, top-level goals (for instance, scoring a soccer goal by a team of players) can be achieved, and how agents should work together in reaching these goals. In particular, it specifies social schemes, a kind of global plans that are stored in the organization, describing which sub-goals should be achieved and in which order to reach the top-level goal. Coherent sets of sub-goals of a scheme, i.e. goals that coherently belong together forming a single task, are grouped together by so-called missions. At the deontic dimension of MOISE+ it is specified to which missions (coherent sets of goals) an agent (playing a certain role) is permitted or obliged to commit. More precisely permissions are of the form $per(r, m, tc)$ in which r is the role an agent plays, m is the mission the agent playing role r may commit to and tc is a time-constraint specifying a set of periods during which this permission is valid, e.g.: every day, from 14h to 16h. Obligations are of the form $obl(r, m, tc)$ meaning that an agent playing role r is obliged to commit to mission m during time period tc. A key aspect of the MOISE+ approach is that obligations and permissions do not refer to (communicative) actions that must (or may) be performed, but to goals as often used in agent programming, i.e. a declarative description of a situation to be achieved. This way an organizational middleware can be better integrated with agent-oriented platforms, as done in J-MOISE+ (Hübner et al., 2007) in which Hübner *et al.* integrate the MOISE+ middleware with the agent-oriented programming language Jason (Bordini et al., 2005). The S-MOISE+/J-MOISE+ middleware informs agents about the missions they are obliged to commit to, when the goals[4] belonging to their missions should be pursued and prevents them (in the sense of regimentation) from committing to missions they do not have permission to. S-MOISE+, however, lacks a monitoring mechanism to detect whether the agents actually fulfill the goals belonging to their obligated missions; it is the agent's own responsibility to inform the middleware about achieved goals. S-MOISE+ is thus somewhat constrained to agents that are benevolent with respect to the organizational goals.

The organizational concepts used by the MOISE+ approach are not underpinned by a formal semantics. In a recent paper (Hübner et al., 2010), however, Hübner and colleagues propose a simple normative language underpinned by an operational semantics (Plotkin, 1981), in which the norms are expressed as conditional rules of the form $\varphi - > \psi$. The consequent of the norm is either an obligation that should be created in case φ holds, or contains a special proposition with predicate $fail$ to denote that the situation described by φ should be regimented. The condition of a rule ranges over facts that describe the situation of the MOISE+ middleware, for example which agents are committed to which missions and which goals are achieved. The idea is that the semantics of many concepts (such as commitment to a goal or a mission) as used in the MOISE+ modeling language can be explained by (automatically) translating them to the normative language. For example, the concept of being committed to a goal can be explained in terms of a (simplified) norm:

$$\begin{gathered} committed(A, M, S) \& goal(M, G, D) \& ready(S, G) - > \\ obligation(A, achieved(S, G, A), \text{`}now' + D) \end{gathered} \tag{2.5}$$

meaning that when agent A is committed to a mission M of scheme S, and mission M includes goal G, and the goal is ready to be pursued next, the agent will be obliged to achieve the goal before its specified deadline D. It is important to note that in (Hübner et al., 2010) it is merely shown that the semantics of various concepts of MOISE+ *can* be explained in this way. No extensive analysis of the semantics of all MOISE+'s concepts is provided.

[4]It should be noted that Jason does not have native support for declarative goals (Hübner et al., 2006a).

2.5 Roles and Organizational Structure

Shakespeare's "all the world is a stage, and all men and women are merely actors" seems to fit well on every day life. During a typical day we naturally play many different roles, e.g. when commuting by train we play the role of passenger, when at work we act in the roles of employee, researcher, author and reviewer, and when shopping on the Internet we take on the role of customer. Etymologically, the word "role" derives from the same word as the English word "roll", namely from the Latin word "rotula" meaning "small wheel". In theater the word role was used to denote the role of papyrus on which the script for an actor was written. This theatrical metaphor, in which actors generate predictable behavior as they play by a script, has been applied by social scientists to explain social systems. Within the field of sociology there seems to be little consensus on what constitutes a role (see (Biddle, 1986) for a survey of the different views). As our interest lies in using the concept of role to build software instead of explaining or simulating social reality, the definition of Britannica online encyclopedia seems to fit our needs best. In Britannica a role is defined as:

> "the behavior expected of an individual who occupies a given social position or status. A role is a comprehensive pattern of behavior that is socially recognized, providing a means of identifying and placing an individual in a society. It also serves as a strategy for coping with recurrent situations and dealing with the roles of others (e.g., parent-child roles). The term, borrowed from theatrical usage, emphasizes the distinction between the actor and the part. A role remains relatively stable even though different people occupy the position: any individual assigned the role of physician, like any actor in the role of Hamlet, is expected to behave in a particular way. An individual may have a unique style, but this is exhibited within the boundaries of the expected behavior.
>
> Role expectations include both actions and qualities: a teacher may be expected not only to deliver lectures, assign homework, and prepare examinations but also to be dedicated, concerned, honest, and responsible. Individuals usually occupy several positions, which may or may not be compatible with one another: one person may be husband, father, artist, and patient, with each role entailing certain obligations, duties, privileges, and rights vis--vis other persons."

2.5.1 The Characteristics of Roles in Computer Science

As we have seen in the introduction, within the area of computer science the concept of role has gained much attention and a myriad of different uses can be found of which some show striking similarities with above description. An extensive overview of the use of roles in the field of computer science has been given by Steimann (2000) who identifies a list of features by which various approaches to the concept of role found in data modeling and object-orientation can be classified. Here, we repeat some of the (possibly mutually exclusive) features listed in (Steimann, 2000) that can also be fruitfully applied for classifying different approaches to roles in the field of multi-agent systems:

1. *An object may play different roles simultaneously.* This is regarded as one of the most broadly accepted properties of the role concept.

2. *An object may play the same role several times, simultaneously.* Just like previous point a fundamental finding. A frequently encountered example is that of an employee holding several employments.

3. *An object may acquire and abandon roles dynamically.*

4. *The sequence in which roles may be acquired and relinquished can be subject to restrictions.* For example, a person can become a teaching assistant only after becoming a student.

5. *Objects of unrelated types can play the same role.*

6. *Roles can play roles.* This mirrors the condition that an employee (which is a role of a person) can be a project leader, which is then a role of the employee (but also another role of the person, although only indirectly). A rather technical subtlety that seems to require that roles are themselves instances.

7. *A role can be transferred from one to another.* It may be useful to let a concrete role dropped by one object be picked up by another, or even to specify the properties of a concrete role without naming a particular role player. For example, the salary of an open position may be specified independently of the person that will be employed.

8. *Roles restrict access.* When addressed in a certain role, features of the object itself (or of other roles of the object) remain invisible.

9. *An object and its roles share identity.* In the object-oriented world this entails that an object and its roles are the same.

Whatever features addressed to a role, the common assumption is that the role metaphor accords better with how we conceive and structure the world around us.

2.5.2 Roles in Multi-Agent Systems

Also within the field of multi-agent systems, that heavily leans on metaphors borrowed from social theories, the role metaphor has been picked up by many researchers. Indeed, roles are a recurring concept in agent development methodologies (for example (Ferber et al., 2003; Odell et al., 2003; Wooldridge et al., 2000; Dignum, 2004)) and they are first-class abstractions in several approaches to implementing computational multi-agent systems (for example (Baldoni et al., 2005), (Baldoni et al., 2007, 2008; Dastani et al., 2004b), and the work of (Esteva et al., 2004), (Hübner et al., 2007)). Just as is the case within the area of computer science, a host of different interpretations of the role concept witnesses the lack of consensus on a crisp definition. The only consensus there seems to be in the area of multi-agent systems is that roles allow us to abstract away from the individuals that will play them. This is a particularly useful characteristic in the development of open systems in which no assumptions can be made about the internals and behavior of the agents that will interact with it. In what follows we give a succinct (non-exhaustive) overview of the different interpretations of the role concept and their use within the area of agent research (appearing in no particular order).

Agent Group Role

In the Agent Group Role (AGR) design model proposed by Ferber and colleagues (2003) a role is conceived as the abstract representation of a functional position of an agent. Roles are used to specify organizations and their structure without committing to particular agents that will enact these roles. Agents always act in a role and can play multiple roles at the same time. Moreover, one role can be played by several agents. In AGR the organization is partitioned in groups to which the roles and agents playing them belong. Communication between agents is limited to the members of a group, i.e. only members of the same group may interact. The organization can be further structured by means of constraints (e.g. cardinality and dependency between roles) and communication protocols describing sequences of messages that flow between roles. In AGR roles do not have a structure; they are merely placeholders for agents used for specifying the organizational structure abstracting away from the agents that will play them.

Roles and organizational rules in the GAIA design methodology

According to the Gaia methodology (Wooldridge et al., 2000) for designing multi-agent systems a role is an abstract description of the expected functionality to be fulfilled by the individual taking up the role. Differently from the AGR model a role has a structure. More specifically, a role is characterized by two types of attributes: 1) permissions describing which resources may or may not be used while carrying out the role; and 2) responsibilities stating the tasks that should be carried out by the player. In Gaia, links between roles are captured by the interaction model, defining the sequences of interactions that may take place between roles.

More elaborate mechanisms for capturing the organizational structure are presented in (Zambonelli et al., 2001a) by Zambonelli *et al.* They propose organizational rules for (amongst others) specifying global constraints and relationships between roles. For example, that two roles cannot be played by the same agent or cardinality constraints stating that a role should be played by a certain number of agents. These organizational rules provide a fine-grained mechanism for specifying the inter-role relationships and constraints.

Roles per AMELI and MOISE+

In many approaches to computational organizational frameworks (cf. the earlier mentioned AMELI framework (Esteva et al., 2004) and the S-MOISE+ framework (Hübner et al., 2006b)) the concept of role is used, comparable to RBAC, as a label to assign normative concepts such as obligations, prohibitions and permissions to agents based on the roles they play. The rationale is the same as in agent design methodologies. That is, we can express the structure and functionality of a multi-agent system abstracting away from the agents that will reside in it. Agents then enact these roles at run-time and all the organizational rules and norms that are associated to the role will be assigned to the agent playing it.

Just as in the design models, various properties of roles can be addressed to further specify the structure of an organization. In AMELI, for example, it is possible to specify incompatibility between roles, meaning that two roles cannot be played by the same agent at the same time. In S-MOISE+ more inter-role relationships can be defined. For example, compatibility, authority, communication, and acquaintance links can be specified between roles to further constrain the organizational structure and the behavior of the agents playing the roles. A compatibility relation between two roles r_1 and r_2 means that an agent playing role r_1 is also allowed to play role r_2. If two roles are connected by a communication link, this implies that agents playing this role may communicate to one another. An authority relationship from a role r_1 to another role r_2 means that the agent playing r_1 has authority over the agent playing role r_2. An acquaintance link from a role r_1 to a role r_2 means that an agent playing role r_1 is allowed to have a representation of the agent playing role r_2[5]. To further constrain the structure of the organization, cardinality constraints may be associated to the roles to define the maximum and minimum number of agents a role is supposed to be played by.

A key aspect of these approaches in specifying organizational constraints, is that they are often defined as hard constraints that cannot be violated by the agents. This pertains to a regimentation of the organizational structure. An exception to this statement is the work of Hübner *et al.* (2010) who propose to define the structural constraints provided by the MOISE+ model in terms of norms (as discussed above). This implies that by using an enforcement strategy

[5]What is meant by having a representation of an agent and having authority over an agent is not defined in (Hübner et al., 2002, 2006b).

of the norms structural constraints may be violated. The interesting idea of using norms to express organizational constraints is illustrated by showing one example of how to express a cardinality constraint. From (Hübner et al., 2010) it is, however, not clear how other constraints such as the incompatibility between roles can be expressed having only obligations. Based on the same idea (which we also proposed in (Tinnemeier et al., 2009a)) we will explore the possibilities of expressing various structural constraints as norms in chapter 5.

Roles for Agent-Oriented Programming

Dastani *et al.* (2003; 2004b) present a richer account of roles in the context of agent-oriented programming. In their formal investigation, a role is attributed mental attitudes that can also be found in agent programming, namely beliefs, goals and planning rules. A role's beliefs specifies the information an agent receives when taking up the role. Its goals specify the states the agent playing the role wants to achieve, e.g. to have bought a good when enacting a buyer role. The planning rules that are typically used for agents to generate plans for reaching their goals and adopting new goals depending on the current situation, are in the context of roles used to express different types of norms. For example, a conditional obligation to pay for received goods can be expressed by a planning rule that generates a goal (or action) to pay if a belief can be derived that the good is received. Agents that enact a role internalize the beliefs, goals and planning rules. By activating a role (initially a role is inactive and only one role can be active at a time) the role's components are executed simultaneously with the agent's own beliefs, goals and planning rules. The role's mental attitudes may thus influence the agent's mental attitudes.

Organizations as Socially Constructed Agents

Another example of attributing mental attitudes to roles can be found in the work of Boella and Van der Torre (2004) who, in modeling an organization, conceive it as a "socially constructed agent". The idea is that agents may describe an organization as if it was an agent having its own goals and beliefs. More specifically, an organization can be decomposed into functional areas, e.g. a university can be decomposed in a direction area and different faculties which each can be further decomposed in departments. Each functional area can be attributed mental attitudes, and at the lowest level we find the roles that can be played by agents, e.g. students, lecturers and the dean. The main advantage of viewing an organization as an agent is that an organization itself can create other social entitities by attributing mental attitudes to them. This way organizations can be hierarchically decomposed; an organization can recursively define sub-organizations and roles by describing them as agents.

Similar to (Dastani et al., 2003, 2004b) roles can also be attributed a mental state, but differently from (Dastani et al., 2003, 2004b) a is linked to the organization. The idea behind considering an organization as an agent is close to the fact that (in the real world) organizations can act as legal entities that can act like agents, albeit via their representative(s) playing roles in the organization. It is, however, important to note that Boella and Van der Torre do not suggest that an organization is actually built in terms of mental constructs:

> "socially constructed agents do not exist, but they are only used in the design analysis to structure an organization. At the end of the process there are only human or software agents which, to coordinate their behavior, behave as if they all attribute the same beliefs, desires and goals to the organization." (Boella and Torre, 2004)

PowerJADE

Building on amongst others powerJava Baldoni *et al.* (2005; 2007) provide an extension of the JADE (Java Agent DEvelopment) framework (Bellifemine et al., 2007) to encompass organizations and roles as first-class abstractions in agent systems (Baldoni et al., 2008, 2009). Like in powerJava a role is defined as an intrinsic part of an organization providing its player powers to act upon the organization. Further, similar to powerJava a power may require certain requirements (or capabilities) from the agent invoking the power. In powerJADE, however, these powers and requirements are not implemented as methods of the class implementing the role, but as JADE behaviors associated to the role. In JADE a behavior defines a task that an agent can carry out and is implemented as an object of a class. The powerJADE middleware provides mechanisms allowing to define protocols for (de-)enacting and (de-)activating roles and allowing agents to invoke powers and powers to invoke requirements in an asynchronous manner.

2.6 On the Notion of Power

Before we end this chapter by providing a general discussion on the matters discussed, it is worth pointing out a notion that has been ubiquitously present throughout this chapter, but has not been mentioned explicitly thus far. It is the notion of power we are referring to here. In general, the concept of power as used within the context of a multi-agent system can be interpreted as the ability of an agent to accomplish its goals at will, cf. (Castelfranchi, 2003).

The first implicit use of the notion of power we encountered in this chapter is explained by Weyns' observation (Weyns et al., 2007) that an environment typically encapsulates resources and provides functionality the agents may use in reaching their goals. By providing actions the agents may perform to physically change the environment (e.g. deleting a file from the computer's hard-drive or moving a physical obstacle by a robot arm in the real world) it gives the agents new powers. This role of power will be the subject of chapter 3.

Not only the environment may give the agents means to achieve their goals, also an organization may provide the agents with new capabilities in the form of institutionalized powers as Jones and Sergot (1996) call them. An example provided by Artikis clarifies the intuition behind the notion of institutionalised power:

> "in an auction house, the auctioneers utterance of the speech act "the item x is sold" counts as, in the auction house, a way of establishing that item x is sold. One then says that the auctioneer has the power, or is empowered, within the auction house, to establish that item x is sold. The same action performed by an agent without this power has no effect." (Artikis, 2003)

What distinguishes institutionalized powers from the powers that are provided by the environment is that "physical" powers allow the agents to change the brute state by providing physical actions (as Jones and Sergot call them), whereas institutionalized powers allow the agents to establish institutional facts (e.g. an item begin sold) that have specific meaning in the organization. Remark the use of "counts as" in Artikis' example. Indeed, Jones and Sergot (1996) use the counts-as rules we discussed before to formalize the notion of institutionalized power. This is the second notion of power we encountered thus far. Remember, that in this dissertation we will not consider constitutive norms.

A third way the concept of power has appeared in this chapter is via the concept of role. The powers (either physical or institutionalized) an agent can exercise usually depend on the roles they play. The role an agent plays may for example appear in the pre-condition of an action or the antecedent of a counts-as rule. By enacting roles, an agent thus obtains new powers. That there are also other, more sophisticated ways through which agents are empowered by the roles they play is demonstrated, for example, by the approach taken by powerJava in which

> "the actions defined for the role in the definition of the institution have access to the state and actions of the institution and to the other roles state and actions: they are powers." (Baldoni et al., 2005)

Also the interpretation of roles we propose in chapter 5 gives the agents new powers.

2.7 Discussion

In this chapter we have discussed the key issues involved in the development of multi-agent systems in which possibly unknown agents dynamically enter and exit and perform their (inter)actions in a shared environment. One of the major issues in the development of such systems is how to coordinate and regulate the behavior of the individual agents in order to guarantee a correct behavior of the system as a whole. We have shown that the exploitation of coordination media having organizational concepts such as roles and norms as first-class abstractions are seen as a potentially powerful candidate to overcome this issue. Indeed, a vast amount of organizational frameworks that are centered around the concept of role and norm have been proposed to date.

Despite the great amount of work in this direction we still observe some major limitations that need to be overcome to fully exploit the potential of organizational frameworks in the development of multi-agent systems. Most of them are related to the gap between agent-oriented research and research in the direction of organization-oriented approaches we mentioned in the introduction of this thesis. In the next chapters in which we aim to overcome these limitations, they will be discussed in more detail. Without getting into too much detail, we observe the following major limitations that will be addressed in this thesis:

1. First and foremost, we observe that none of the approaches to organization-oriented multi-agent systems is underpinned by a formal operational semantics (Plotkin, 1981), whereas many agent-oriented approaches are. An exception is our own work (Dastani et al., 2008, 2009) and the work of Hübner *et al.* who provide an operational semantics of some parts of the MOISE+ framework (Hübner et al., 2010). An operational semantics allows us to explain the meaning of programming constructs in a rigorous manner. The importance of this is, for example, demonstrated by the study of Van Riemsdijk *et al.* (2010) who tried to capture the constructs provided by the MOISE+ framework in a formalism. The study revealed unclarity of some concepts, for which we blame the impreciseness of a natural language description. Moreover, an operational semantics allows us to formally prove some main properties the constructs exhibit. Such properties can then be compared with properties from, for example, logical frameworks to validate our proposal. All concepts proposed throughout this thesis will be underpinned by an operational semantics.

2. Organizational frameworks are an important part of the agents' environment. Most existing organizational frameworks, however, are merely used for regulating the agents' interactions by means of normative concepts, whereas some others are used for structuring the social relationships between the agents in terms of groups and roles. As argued by Weyns and colleagues (2007) an environment should also (amongst other) be able to encapsulate resources and provide functionality the agents may use in achieving their objectives. In other words, it should give the agents powers to physically change their environment. In chapter 3 we offer a solution in which an organizational artifact may also be used to encapsulate resources and functionality the agents may use. In chapter 5 we provide a mechanism by which agents may obtain even more sophisticated powers by enacting and playing roles.

3. In many approaches to organizational frameworks the notion of norm is one that relates to (communicative) actions an agent should or should not perform, rather than to a declarative description of a state that should or should not be achieved. As we shall argue in chapter 4 this does not accord well with the concept of declarative goal that is often used in agent-programming languages. In chapter 4 we develop a solution that takes a declarative view on norms.

4. The dynamics of the concepts of role and norm is intimately related with the structure of the concepts. Agents, for example, may dynamically enact and de-enact roles at run-time. Moreover, deontic concepts such as obligations and permissions are dynamically created. However, despite the observation that a static view in which the specification of the norms is defined at design time and cannot be modified at runtime does often not suffice (Castelfranchi, 2000; Boella et al., 2006; Dignum, 2009b), little work has been done up to now to support the run-time modification of the norms. Being able to dynamically adjust the set of norms that has been put in place at design-time allows us to better cope with the actuality of the running system resulting from the exact nature of the agents' behavior. In chapter 6 we develop a solution to overcome this issue. Because the issues involved w.r.t. computational norm change are better explained after having presented our own representation of norms, a discussion thereof as well as a discussion of organizational frameworks that do support run-time norm change is postponed to chapter 6.

5. There are many different interpretations of the concept of role, and a crisp definition is still lacking. Most organizational frameworks take a very simple view in which a role is merely a label, but other approaches have shown that a role can be a computational entity with its own state and behavior. Research from the direction of agent-oriented programming has even shown that roles may be constructed by agent-oriented concepts such that agents can reason with them. In chapter 5 we will use the approaches to roles discussed in this chapter to identify the key properties the concept of role should exhibit for efficiently constructing agent organizations and reducing the gap between agent concepts and organizational concepts.

6. The organizational structure is to a large extent encompassed by various constraints on the roles, e.g. a constraint that a role cannot be played by the same individual at the same time. In current approaches such constraints are typically expressed as hard-constraints that cannot be deviated from (cf. regimentation). Such an approach limits the agents' autonomy and also the flexibility of the organizational artifact. In chapter 5 we investigate to what extent the violable norms in chapter 4 are suitable for expressing these constraints.

7. Agents are faced with choices about what to do next, e.g. executing a plan to reach a goal or updating beliefs on the basis of incoming percepts. The best order in which agents perform such activities is application-dependent and is encoded by their deliberation cycle. An organizational artifact is faced with comparable decisions about how to perform its activities, e.g. enforcing all norms versus enforcing only some in time critical systems. Indeed, in (Aştefănoaei et al., 2009a) different strategies are discussed for enforcing a set of norms. Inspired by the notion of a deliberation cycle we introduce a coordination cycle (in chapter 3), by which an organizational artifact determines what to do next.

Chapter 3

Setting the Scene: Organizational Artifacts

It has been variously stressed that the environment plays an important role in the engineering of open multi-agent systems, see for example (Weyns et al., 2007; Ricci et al., 2006, 2007). An environment is a first-class citizen encapsulating its own domain-specific functionality and is developed independently from the agents that reside in the multi-agent system. This functionality provided by the environment often includes organizational infrastructures for coordinating the agents' interactions. This accords with the widely promoted organization-centric view on multi-agent systems, cf. (Jennings, 2000; Zambonelli et al., 2001b; Ferber et al., 2003; Grossi, 2007; Dignum, 2009a), in which a multi-agent system is designed and analyzed using an explicit notion of social concepts such as roles and norms.

In this chapter we set the scene for building organization-centric multi-agent systems by introducing the notion of organizational artifact (previously introduced in (Dastani et al., 2008) and further discussed in (Dastani et al., 2009; Tinnemeier et al., 2009a,b,c, 2010)); a computational entity that is deployed in the environment of a multi-agent system and implements the organizational aspect. In particular, the main contributions of this chapter are as follows:

- We introduce our view on organization-oriented multi-agent systems, and explain the role of organizational artifacts in constructing them in section 3.2. Contrasting most existing organizational frameworks (see chapter 2 for some examples) our notion of organizational artifact is able to encapsulate resources and provide functionality the agents may use in achieving their objectives.

- We elaborate upon the elementary programming constructs by which an organizational artifact can be programmed in section 3.3. The constructs that will be presented in this chapter are essential for programming any organizational artifact. In later chapters we extend this language by additional constructs that can be optionally used depending on the problem at hand.

- We discuss the intuitive meaning of the programming constructs and illustrate how they can be used by an example involving a conference management system. We introduce this example in section 3.1 and will use it throughout the rest of the thesis in explaining the additional concepts that are introduced in later chapters.
- A formal semantics of the programming constructs is essential for a thorough understanding of the programming language at hand. In section 3.4 we explain the meaning of the introduced constructs by means of an operational semantics. Special attention is given to the concept of organizational coordination cycle (comparable to an agent's deliberation cycle) that explains the order in which different programming constructs are applied by an organizational artifact.
- Organizational artifacts pertain to the environmental dimension of multi-agent systems. In the previous chapter we discussed the responsibilities assigned to environments by Weyns and colleagues (2007). In that chapter we also discussed Davidsson's different degrees of openness by which multi-agent systems can be categorized. In section 3.5 we discuss to what extent our notion of organizational artifact satisfies the responsibilities assigned to environments and how they can be deployed for programming the different types of openness. In that section we also relate our work to other approaches to programming organization-oriented multi-agent systems.

In summary, by introducing the concepts underlying an organizational artifact and the programming language by which we can define them, we introduce the pillars for the organizational programming language introduced throughout this thesis.

3.1 Conference Management System Example

To illustrate our approach of programming agent organizations we use an example involving a conference management system. Essentially, the conference management system consists of two related parts, namely the reviewing system and the registration system. The reviewing system has as aim to guide agents playing the role of reviewer, chair and author in the process of uploading and reviewing papers. The registration system is responsible for managing the registration of conference attendants and their payment. A similar example has been used by Zambonelli *et al.* (2003) in explaining the Gaia methodology. They argue that a conference management system naturally touches upon many interesting features that are characterizing for multi-agent systems. For example, authors that submit their papers are not known at design time and start their interactions dynamically at runtime. This pertains to the openness of the system. The activities of authors and reviewers typically take place in a distributed manner. Moreover, authors and reviewers each have their own agenda, possibly conflicting with those of other agents participating in the system. This may result in opportunistic and unpredictable behavior. These characteristics and the fact that a conference management system needs little explanation in the face of our audience, motivate our choice for this example.

The reviewing system goes through different phases in which activities take place. Initially, the system is closed. Opening the system will put it in the phase in which abstracts can be uploaded. Authors should first register their abstracts before papers can be uploaded. Uploading an abstract will assign an author a unique number by which papers are identified. After the abstract phase comes the submission phase in which authors can upload their contributions. After the submission phase ends, the review phase starts in which the chair assigns reviewers papers to review. During this phase reviewers can upload their reviews. Reviews are collected and decisions are made about which papers will be accepted in the collect phase. When decisions have been made for each paper, the notification phase starts in which the authors are notified. Finally, when all authors are notified, the registration phase starts in which authors of accepted papers are expected to register for the conference. For this example we assume that only the chair can cycle between different phases.

Just like the reviewing system, the registration system goes through different phases. It is closed initially, meaning that registration is not yet possible. The system is automatically opened when the reviewing system is put in the notification phase. Opening the system will start the early registration phase in which the participation fee is lower than the normal fee to be paid in the registration phase that comes after the early registration. Finally, registration is no longer possible when the system is closed, e.g. some time prior to the conference or when all tickets are sold out. It should be noted that the early registration phase of the registration system is in sync with the notification phase of the reviewing artifact and that the registration phase of the registration system is in sync with the registration phase of the reviewing system. All phases the conference management system can be in and their transitions are summarized in figure 3.1.

3.2 Organizational Artifacts

In this section we explain our view on organization-oriented multi-agent systems. In particular, we introduce the notion of organizational artifact and their core building blocks. Moreover, we discuss the need for an organizational coordination cycle that explains the activities an organizational artifact employs as a result of the agents' interactions.

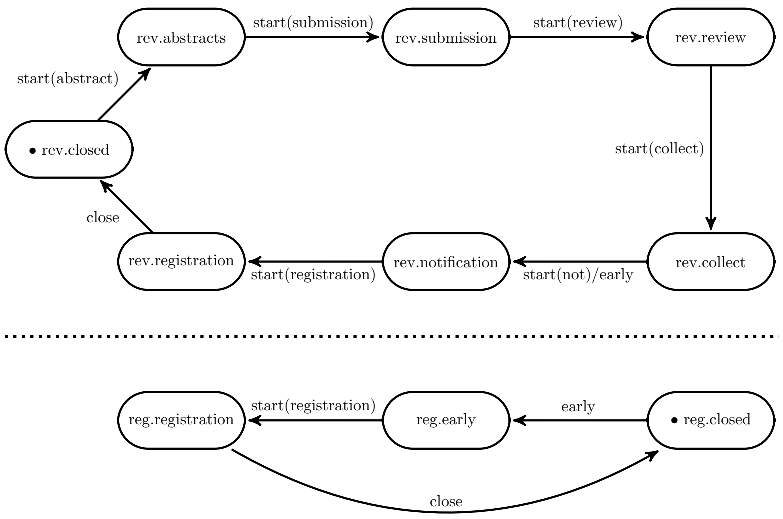


Figure 3.1: The different phases the conference management system goes through depicted by a state chart. The system is composed of two (synchronized) systems that run in parallel (as denoted by the dotted line). All the phases of the reviewing system are shown above of the dotted line, and that of the registration system below. A dot in a state denotes an initial state.

Multi-agent Systems and Organizational Artifacts

A multi-agent system, as we conceive it, is composed of a collection of agents and a collection of organizational artifacts as illustrated by figure 3.2. The agents are heterogeneous, meaning that they are possibly implemented by different parties using different programming languages. To reach their goals, agents interact with each other and may exploit the functionality provided by the organizational artifacts. To explain the intuition of an organizational artifact recall the definition of an artifact as coined by Ricci *et al.* who define an artifact as:

> "a computational device populating agents' environment, designed to provide some kind of function or service, to be used by agents – either individually or collectively – to achieve their goals and to support their tasks." (Ricci et al., 2006)

Our notion of an organizational artifact can be seen as a specialization of an artifact as defined in (Ricci et al., 2006) with the additional purpose to regulate the agents' interactions. We (informally) define an organizational artifact as:

> "An organizational artifact is an artifact that uses an explicit notion of organizational abstractions to coordinate and regulate the agents' behavior."

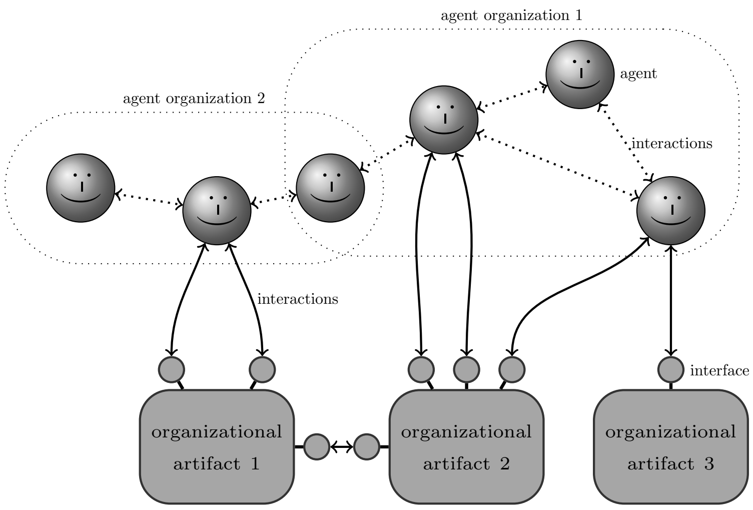


Figure 3.2: Conceptual representation of a multiagent system consisting of a heterogeneous collection of agents interacting with organizational artifacts. The interactions between agents consist of communication messages, the interactions between agent and artifact of messages (from artifact to agent) and actions (from agent to artifact), and the interactions between artifacts consist of actions solely.

Organizational Artifact Building Blocks

A (simplified) conceptual representation of the internals of an organizational artifact is depicted by figure 3.3. As can be seen from this figure some parts of an organizational artifact are mandatory, whereas other parts can be optionally added depending on the specific application the artifact is deployed for. This allows for a greater flexibility in choosing the right abstractions for a particular problem domain. Each organizational artifact encapsulates a domain-specific state and function, e.g. a database and accompanying operations to store information about submitted papers for the conference management system example. The domain-specific state is modeled by a set of *brute facts*, taken from Searle (1995). These facts store, for instance, information about which papers have been uploaded and which phase the system is in. The artifact's functionality that agents may exploit to reach their goals is provided by actions that can be invoked by the interacting agents. Agents can perform actions to modify the brute state, e.g. for uploading a paper or a review and actions for communicating with other agents interacting with the organizational artifact, e.g. the chair may inform each reviewer about which papers to review. The effects actions have on the brute state are described by so-called *effect rules*. Effect rules specify which brute facts should be asserted or retracted as a result of performing an action. Effect rules thus empower (Castelfranchi, 2003) the agents to alter the artifact's

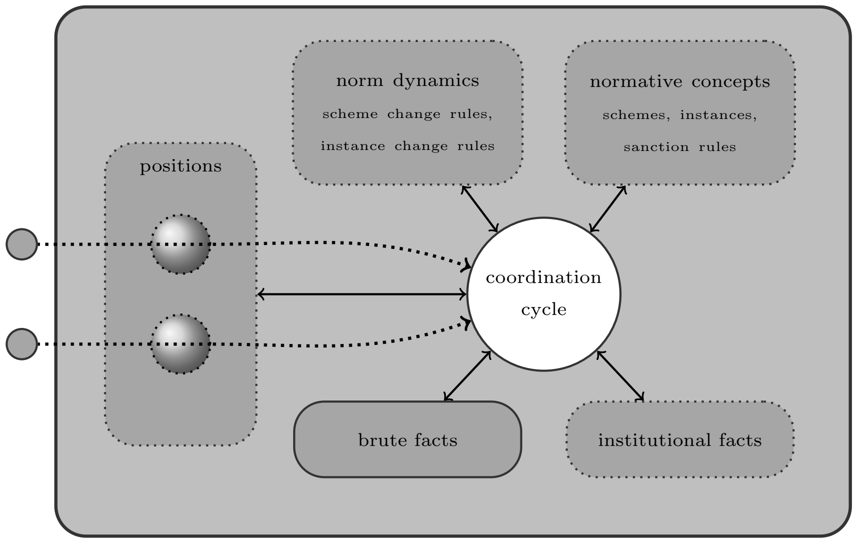


Figure 3.3: A (simplified) conceptual representation of the internals of an organizational artifact. Optional concepts (e.g. norms, positions) are outlined with a dashed border. Solid arrows denote the reading and modification of the concepts as explained in more detail below. Dotted arrows denote actions and messages between agent and artifact.

physical state as described by the brute facts.

The operations by which agents may change the internal state of the artifact may also have side effects in the form of communication messages sent to other agents and actions performed upon other artifacts. By allowing an artifact to send communication messages to other agents, the artifact can function as a communication medium. That is, agents can send messages to other agents without needing to know their identity. Moreover, the artifact can provide agents feedback on the result of their actions by sending them a message. Allowing artifacts to act upon each other enables information sharing across different artifacts; by performing an action an artifact may automatically update the status of another artifact upon a change of its own state. It is important to emphasize that the side effects of sending messages and executing actions only occur as a consequence of an agent performing an action upon the artifact. An artifact merely reacts to agents' actions. It does not perform actions and does not send messages on its own initiative. Organizational artifacts thus not have a high level of autonomy, which distinguishes them from agents.

From the artifact's point of view, agents are treated as black boxes, i.e. no assumptions are made about their internals and no control can be exerted over their internal state. Although our approach is specifically tailored to BDI-oriented agents, a multi-agent system may also include active entities that implement tasks and processes that are better implemented by a traditional software engineering approach rather than an agent-oriented approach, e.g. a dedicated process that performs an action updating the state of an artifact at settled times.

When little can be assumed about the internals of the agents interacting with an artifact and when these agents are built by (unknown) developers different from the artifact's developer this implies that also no assumptions can be made about their behavior. In such a case normative concepts may be put into place to guide the agents in interacting with the organizational artifact in a meaningful way. These normative concepts include *norm schemes* that specify the circumstances under which *obligations* and *prohibitions* (*norm instances*) are created. *Sanctioning rules* specify the consequences of violating or abiding by a norm, i.e. they can be used to punish agents who do not abide by the norms and reward those who do. The normative component will not be discussed in this chapter, but is explained in detail in chapter 4. Because in many cases a view in which the norms are specified at design-time and cannot be changed at run-time is not sufficient when dealing with the unpredictable outcomes of the agents' interactions, we introduce programming constructs for changing the norms at run-time. These constructs include mechanisms for changing the norm schemes and constructs for changing the obligations and prohibitions that arise from them. Just like the normative component, these constructs will not be discussed in this chapter.

In case the set of agents that will interact with the organization is not fixed at design time (the system is not closed (Davidsson, 2001)) and not a priori known, i.e. when unknown agents dynamically start and stop interacting with the organizational artifact, *roles* may be introduced. Roles allow the artifact's developer to abstract from the individuals that will play them. Norms, for example, can then be associated to roles that agents play instead of associating them to the agents directly. In this chapter we adopt the simple view in which roles are merely labels denoting a role that can be played by an agent. In chapter 5 a richer view on roles is introduced in which positions (roles played by an agent) act as organizational representatives of the agents playing them and provide functionality to their players. Note that this view accords with the semi-closed (Davidsson, 2001) perspective on a multi-agent system. When positions are deployed agents no longer perform actions to change the brute state, but instead interact with the positions they occupy empowering the agents to modify the brute state through their positions.

Information about which roles agents have enacted, and violations and fulfillment of norms is stored by the institutional facts, again taken from Searle (1995). Institutional facts store information about the organization specific state of the artifact, as opposed to brute facts which store information about the domain-specific state of the artifact. The fact that a role named r has been enacted by an agent i is stored as an institutional fact $rea(i, r)$. The fact that a norm has been violated is stored by an institutional fact with predicate $viol$, whereas the obedience of a norm (e.g. fulfilling an obligation) is denoted by a fact with predicate symbol *obey*. These institutional facts will, however, not be discussed further in this chapter.

Just like an agent needs to weigh executing its plans for reaching its goals against rethinking its plans, goals and beliefs in a continuously changing environment, an organizational artifact has to do a similar weighing. It should, for example, seek a balance between enforcing the norms and changing them in the light of unpredictable interactions of agents. Similar to an agents deliberation cycle, we define the concept of organizational *coordination cycle*. A coordination cycle defines what the organizational artifact should do next, e.g. determining the effect of an action and changing the brute state accordingly, handling role enactments, determining which norms are complied with or violated, applying sanctions, and changing the norms based on the current situation. In this chapter we will focus at two particular steps the organizational coordination cycle encompasses, namely determining the effect of an action on the brute state and handling the enactment and de-enactment of roles. These steps and the concepts of the artifact that are involved are depicted in figure 3.4 and 3.5. In these pictures that will be used throughout this thesis, arrows from coordination process (depicted as circle) to store (depicted as rounded box) denote the modification of the store's elements, whereas an arrow in the opposite direction denotes the reading of the store's elements. Dotted arrows pertain to agent-artifact or artifact-artifact interactions.

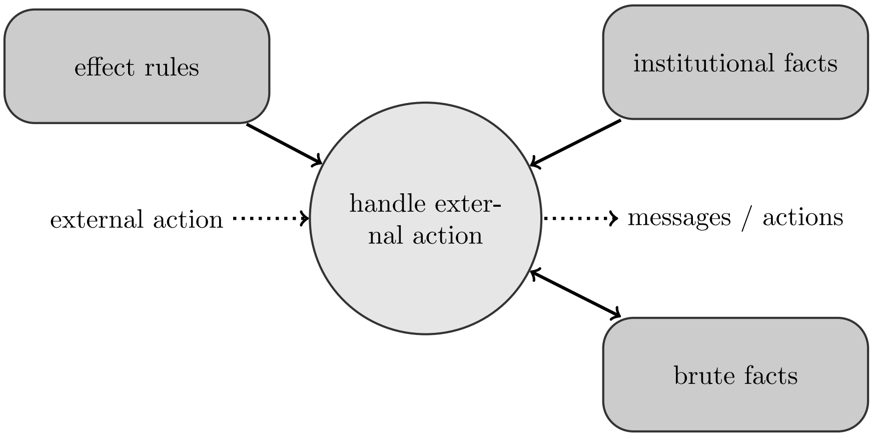


Figure 3.4: The effect of an external action is determined by the effect rules that specify the facts that should be accommodated to the brute state given a situation described by the brute and institutional facts. The performance of an action may result in messages to be sent and actions to be performed.

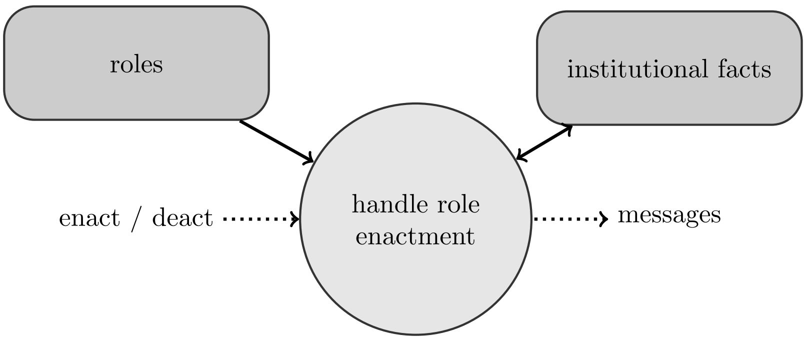


Figure 3.5: An agent i enacting a role r results in the assertion of a fact $rea(i, r)$ to the institutional facts. De-enacting this role retracts this fact. By sending a message the agent is notified in case of failure to (de-)enact a role.

```
⟨artifact⟩ = "Name:" ⟨id⟩ [ ⟨roles⟩ ] [ ⟨facts⟩ ] [ ⟨effects⟩ ];
⟨roles⟩    = "Roles:" ⟨role⟩ { ⟨role⟩ };
⟨facts⟩    = "Facts:" ⟨b − atom⟩ { ⟨b − atom⟩ };
⟨effects⟩  = "Effects:" ⟨effect⟩ { ⟨effect⟩ };
⟨effect⟩   = "{" ⟨pre⟩ "}" ⟨action⟩ "{" ⟨post⟩ "}";
⟨pre⟩      = ⟨b − lit⟩ | ⟨r − lit⟩ | ⟨pre⟩ { "," ⟨pre⟩ };
⟨post⟩     = ⟨b − lit⟩ | ⟨send⟩ | ⟨do⟩ | ⟨post⟩ { "," ⟨post⟩ };
⟨b − lit⟩  = "true" | ⟨b − atom⟩ | "not" ⟨b − atom⟩;
⟨r − lit⟩  = ⟨r − atom⟩ | "not" ⟨r − atom⟩;
```

Figure 3.6: EBNF grammar of normative artifacts.

3.3 Programming Organizational Artifacts

This section explains the basics of the programming language by which the core ingredients of an organizational artifact can be programmed. To specify the initial configuration of a basic organizational artifact is to specify the roles that can be played, its initial brute state, the actions that can be invoked by agents and optionally the roles that can be played by agents. The EBNF syntax by which our programming language is specified is listed in figure 3.6. The basic elements our programming language is built of are explained in table 3.1. In what follows all programming constructs and their intuitive meaning will be explained by means of our conference management system example.

The conference management system is implemented by two mutually interacting artifacts, one implementing the reviewing system identified by `rev` and one implementing the registration system identified by `reg`. The code specifying the first is shown in code fragment 3.2 and the code for the latter is shown in code

⟨id⟩	a term uniquely identifying an artifact.
⟨role⟩	a term identifying a role that can be played.
⟨b−atom⟩	a first-order atom denoting a brute fact. The special facts starting with predicate symbol *viol*, *obey* and *rea* (their meaning to be explained later on) are excluded from the set of brute facts.
⟨r − atom⟩	a first-order atom of the form $rea(i, r)$ in which r denotes a role and i the agent playing it; "rea" is short for role enacting agent. All rea facts are institutional facts.
⟨action⟩	a first-order atom of the form $P(t_1, \ldots, t_n)$ in which predicate P denotes the name of the action, and $t_1, \ldots, t_n$ the action's arguments.
⟨send⟩	a first-order atom of the form $Send(r, p, c)$ in which r denotes the message's receiver, p its performative and c its content.
⟨do⟩	a first-order atom of the form $Do(id, \alpha)$ in which id denotes the artifact upon which action α is to be performed.

Table 3.1: Elementary syntactical constructs.

fragment 3.1.

To interact with the artifact agents enact roles. The roles that are relevant for the registration system are specified on line 2. The only role that can be played is the role of author. Roles that can be played in interacting with the reviewing artifact are the roles of author, reviewer and chair as specified on line 2 of code fragment 3.2.

The domain-specific state of an organizational artifact is described by the brute facts. Brute facts are composed of ground first-order atoms, i.e. are free of variables. The initial brute state of the artifact implementing the registration system specifies that the system is initially closed, the early registration fee amounts to EUR 300,- and the normal fee EUR 350,-. This is shown on line 3 of code fragment 3.1. The initial brute state of the artifact implementing the reviewing system is shown on line 3 of code fragment 3.2. Initially the system is closed. The fact `id` is used to remember the last unique identifier that has been assigned to a paper. Initially, this identifier is set to zero. As at this point no papers or reviews have been uploaded to the system, no facts about this are specified.

The brute facts change under the performance of actions by external parties. The effects describe which effect an action has on the brute state and are used by the organizational artifact to determine the resulting brute state after performance of the action. They are defined by triples of the form (Φ, α, Ψ), intuitively meaning that when action α is executed and set of facts Φ is derivable by the current brute state, the set of facts denoted by Ψ is to be effectuated in it. The first argument of action α is always the agent performing it. The precondition may contain brute facts as well as institutional facts specifying the enactment of roles by agents. This way actions that may be performed can be related to roles played by agents instead of relating them to agents directly. The actions of the reviewing artifact are shown on lines 4 - 50 of code fragment 3.2. Recall that a

Code fragment 3.1 Code implementing the registration system.

```
Name: reg                                                                       1
Roles: participant                                                              2
Facts: phase(closed) fee(early,300) fee(normal,350)                             3
Effects:                                                                        4
{phase(closed)}                                                                 5
  start(rev,early)                                                              6
{not phase(closed), phase(early)}                                               7
                                                                                8
{phase(early)}                                                                  9
  start(rev,registration)                                                      10
{not phase(early), phase(registration)}                                        11
                                                                               12
{phase(registration)}                                                          13
  close(rev)                                                                   14
{not phase(registration), phase(closed)}                                       15
                                                                               16
{phase(early)}                                                                 17
  register(rev,P,author)                                                       18
{registration(P,author,uncomplete)}                                            19
                                                                               20
{rea(P,participant)}                                                           21
  register(P,P,participant)                                                    22
{registration(P,participant,uncomplete)}                                       23
                                                                               24
{rea(P,participant), phase(early), registration(P,T,uncomplete),              25
 fee(early,Amount)}                                                            26
  pay(P,Amount)                                                                27
{not registration(P,T,uncomplete), registration(P,T,complete)}                 28
                                                                               29
{rea(P,participant), phase(registration), registration(P,T,uncomplete),30
 fee(normal,Amount)}                                                           31
  pay(P,Amount)                                                                32
{not registration(P,T,uncomplete), registration(P,T,complete)}                 33
```

fact `rea(i,chair)` denotes that an agent identified by `i` has enacted the role of chair. The first effect rule specifies that an agent playing the role of chair can open the reviewing system by performing a `start` action. After successful performance, that is, if the system is still closed at the time of the action execution, the system will be no longer closed and will be in the phase in which abstracts can be uploaded. The actions for switching to other phases are all defined in a similar fashion. After the specification of all the actions to move between the different phases follows the specification of actions to invite reviewers, upload abstracts and papers, (re)assign papers to reviewers, upload reviews, accept or reject papers, update the registration status of authors of accepted papers, and to notify authors. Note how the precondition is used to relate these actions to a certain phase. The actions that can be used to act upon the registration system include switching between different phases and paying. These are defined on lines 4-33 of code fragment 3.1. Note that the registration is final only if the payment has been received.

Thus far, we only introduced means for agents to modify the brute state of the artifact. As we already mentioned, interaction between artifact and agent is not limited to only agents performing actions upon the artifact. In fact, an artifact can communicate with agents by means of sending communication messages, and an artifact can even perform actions upon other artifacts. This is achieved through designated atoms which may be used in the postcondition of an effect specification, and which will not be added to the brute facts, but have side effects instead. In particular, two types of special atoms are introduced. One for interacting with agents by sending communication messages and one for performing actions upon other (organizational) artifacts in the environment of the artifact.

To enable an artifact to send communication messages we introduce a ternary atom, `Send`, which takes as first argument the receiver of the message, as second argument the performative (e.g. inform, request, etc.) and as third argument the content of the message. The format of a send atom is intentionally kept simple. Extending it to, for example, a FIPA [1] compliant representation does not raise any technical difficulties. An example of the usage of such a communicative act can be found in the effect specification defining the effect of the action for notifying authors. Performance of this action will have as effect that a message will be sent informing the author about the acceptance or rejection of the submitted paper. Providing constructs to let the artifact send communication messages not only enables the artifact to be used as a communication medium coordinating the agent's interactions, but it also gives a means to provide an agent performing an action feedback about the result of the performance. Consider, for instance, the action of the reviewing system for uploading an abstract. Performing this action will assign a unique identifier the agent should use in further interactions to identify its paper. Therefore, we want to return the assigned identifier to the author after having successfully uploaded an abstract.

As can be seen from figure 3.2, artifacts interact both with agents and other artifacts. Interaction with agents is achieved through communication messages, whereas interaction with other artifacts is achieved by means of performing actions. To allow an artifact to act upon other artifacts we introduce a binary atom, `Do`, with as first argument the identifier of the artifact to fire the action upon and as second argument a function term denoting the action with its parameters. An example of such an action can be found in the effect specification for the action to accept a paper as listed in code fragment 3.2. Because authors of accepted papers are obliged to register for the conference, the registration system needs to know the authors of accepted papers. Therefore, the action of accepting a paper has as side effect that the registration artifact's `register` action is executed, which registers the author. Another example is the action for closing the reviewing system (the second effect rule). Performing this action automatically closes the registration system also. This way the phases of the registration and reviewing artifact are synchronized.

[1] see http://www.fipa.org

Code fragment 3.2 Code implementing the reviewing system.

```
Name: rev
Roles: chair, author, reviewer
Facts: phase(closed), id(0)
Effects:
{rea(C,chair), phase(closed)}
  start(C,abstract)
{not phase(closed), phase(abstracts)}
  :
  :
{rea(C,chair), phase(registration)}
  close(C)
{not phase(registration), phase(closed), Do(reg,close())}

{rea(C,chair)} invite(C,R) {invited(R)}

{rea(A,author), phase(abstracts), id(PId)}
  uploadAbstract(A)
{abstract(A,PId), not id(PId), id(PId+1), Send(A,inform,id(PId))}

{rea(A,author), phase(submission), abstract(A,PId)}
  uploadPaper(A,PId)
{paper(A,PId)}

{rea(C,chair), phase(review), paper(A,PId)}
  assign(C,R,PId)
{assigned(R,PId)}

{rea(C,chair), assigned(R1,PId)}
  reassign(C,R1,PId,R2)
{not assigned(R1,PId), reassigned(R1,PId,R2)}

{rea(R,reviewer), phase(review), assigned(R,PId)}
  uploadReview(R,PId,Decision)
{review(R,PId,Decision)}

{rea(C,chair), phase(notification), paper(A,PId)}
  accept(C,PId)
{accepted(PId), Do(reg,register(A,author))}

{rea(C,chair), phase(notification), paper(A,PId)}
  reject(C,PId)
{not accepted(PId)}

{rea(C,chair), paper(A,PId), accepted(PId)}
  notify(C,A,PId)
{Send(A,inform,accepted(PId))}

{rea(C,chair), paper(A,PId), not accepted(PId)}
  notify(C,A,PId)
{Send(A,inform,rejected(PId))}
```

3.4 Executing Organizational Artifacts

Having defined the syntax and intuitive semantics of the programming constructs by which organizational artifacts can be specified, we now turn to the formal semantics of these constructs. We start with some preliminary definitions and functions to ease our notation.

3.4.1 Preliminaries

Throughout this chapter and the rest of this dissertation we use some auxiliary functions in defining the semantics of our programming language. Our language heavily builds on brute and institutional facts that are described by first-order atoms (typically denoted by ϕ) of the form $P(t_1, ..., t_n)$ with P a predicate symbol and $t_1, ..., t_n$ either a constant, variable or function term of the form $f(t_1, ..., t_n)$. We introduce a first-order entailment relation to assess whether formula's are derivable from a set of (brute or institutional) facts. The clause dealing with conjunctions will not be used in this chapter, but it will be needed in subsequent chapters. As substitutions are essential in defining this entailment relation, we define them first.

Definition 3.1 (Substitution and unification) *A substitution, typically denoted by θ, is a finite and possibly empty set of pairs $\{x_0/t_0, ..., x_n/t_n\}$ in which t_i denotes a term and x_i denotes a variable for $0 \leq i \leq n$.*

The application of a substitution θ to a constant term c, variable term x, function term $f(t_1, ..., t_n)$ or atomic formula $P(t_1, ..., t_n)$ is defined as follows:

- $c\theta = c$
- $x\theta = t\theta$ *if* $x/t \in \theta$, *otherwise* $x\theta = x$
- $f(t_1, ..., t_n)\theta = f(t_1\theta, ..., t_n\theta)$
- $P(t_1, ..., t_n)\theta = P(t_1\theta, ..., t_n\theta)$

Moreover, given first-order atoms ϕ_1, ϕ_2, we define a function unify(ϕ_1, ϕ_2) that evaluates to the smallest substitution such that $\phi_1\theta = \phi_2\theta$, and $\perp$ if such a substitution does not exist.

Definition 3.2 (Entailment Operator) *Given a set of ground, first-order atoms X, (brute or institutional) first-order atom ϕ, (conjunctions of) first-order atoms $\psi_1(\overline{x})$ and $\psi_2(\overline{y})$ with sets of variables $\overline{x}, \overline{y}$ occurring in them, and substitution θ. The entailment operator $\models$ is defined in the following manner:*

- $X \models \phi$ *iff there exists a ground substitution θ such that $\phi\theta \in X$*
- $X \models \text{ not } \phi$ *iff there is no ground substitution θ such that $\phi\theta \in X$*
- $X \models (\psi_1(\overline{x}) \texttt{ and } \psi_2(\overline{y}))\theta$ iff $\exists\theta_1 : [\theta_1 = \theta|\overline{x}$ and $X \models \psi_1(\overline{x})\theta_1$ and $\exists\theta_2 : [\theta_2 = \theta|(\overline{y} \setminus \overline{x})$ and $X \models \psi_2(\overline{y})\theta_1\theta_2]]$

We also extend the entailment operator to deal with a set of literals (i.e. a set of positive and negative atoms) Φ:

- $X \models \Phi$ *iff* $\forall\phi \in \Phi : X \models \phi$

The (brute and institutional) facts evolve under the performance of actions. In defining the semantics of our programming language we need to update these sets of facts. For this purpose we introduce an update operator. Essentially, this update operator updates a set of atoms X with a set of literals Y by adding all positive literals from Y to X and removing all negative literals contained in Y from X. This is defined by the following definition.

Definition 3.3 (Update Operator) *Let $NegLit(\phi)$ and $PosLit(\phi)$ be special predicates denoting, respectively that a first-order literal is preceded by a negation (i.e.* `not`*) or not. Given a set of ground, first-order atoms X and a set of ground, first-order literals Y we define the update operator $\uplus$ as follows:*

$$X \uplus Y = (X \setminus \{\phi \mid \phi \in Y \text{ and } NegLit(\phi)\}) \cup \{\phi \mid \phi \in Y \text{ and } PosLit(\phi)\}$$

We endow the syntax by which organizational artifacts can be programmed with an operational semantics (Plotkin, 1981). In operational semantics the behavior of a programming language is described in terms of transitions between program configurations. A configuration describes a state of the program and a transition is a transformation of one configuration γ into another configuration γ', denoted by $\gamma \rightarrow \gamma'$. The transitions that can be derived for a programming language are defined by a set of derivation rules of the form $\frac{P}{\gamma \rightarrow \gamma'}$ with the intuitive reading that transition $\gamma \rightarrow \gamma'$ can be derived in case premise P holds. An execution trace in a transition system is then a sequence of configurations that can be generated by applying transition rules to an initial configuration. An execution thus shows a possible behavior of the system at hand. All possible executions for an initial configuration show the complete behavior. The notion of an execution trace is formally defined below.

Definition 3.4 (Execution Trace) *An execution in a transition system $\mathcal{T}$ is a sequence of transitions $\gamma^0 \rightarrow \gamma^1 \rightarrow \ldots \rightarrow \gamma^n$ (often written as $\gamma^0 \rightarrow^* \gamma^n$) such that for each $0 \leq i < n$ a derivation $\gamma_i \rightarrow \gamma_{i+1}$ can be made in $\mathcal{T}$.*

In this chapter we are concerned with three types of configurations, namely an agent configuration that defines an individual agent, an organizational artifact, and a multi-agent system that is composed of a set of agents and a collection of organizational artifacts they interact with. The transitions operating on these configurations are defined by three different types of transitions each defined by their own transition system, viz. the agent transitions which will be denoted by $\rightarrow_{agt}$, artifact transitions denoted by $\rightarrow_{org}$, and transitions at the multi-agent level which we denote by $\rightarrow_{mas}$. The transition system that defines the execution of the normative multi-agent system as a whole is defined compositionally in terms of agent transitions and artifact transitions. Next, we define the transitions that are found at the agent level, artifact level and the multi-agent level.

3.4.2 Agent Level Transitions

Because in our view no assumptions about the internals of the individual agents are made, definition of their configuration and transition system that explains their execution are not provided here. Examples of such definitions can be found in, for example (Dastani, 2008; de Boer et al., 2007; Rao, 1996). We do assume, however, that agents are capable of performing observable actions to interact with the organizational artifact and are able to receive messages sent by other agents or artifacts. In particular, an agent can perform external actions to modify the brute state of the organization. As previously mentioned, in specifying the effects of the action we assume these actions to be of the form $P(i, t_1, ..., t_n)$ in which the predicate name P denotes the name of the action and the first term i corresponding with the unique identifier of the agent performing the action. At the multi-agent level, however, we also need to know which artifact j this action is action is performed upon. Therefore, at this level actions are of the form $P(i, j, t_1, ..., t_n)$. Besides performing observable external actions, agents are also assumed to perform non-observable actions that only modify their internal state. In concrete, the following transitions are assumed for an agent configuration A identified by i:

1)	$A \overset{P(i,j,t_1,...,t_n)!}{\longrightarrow_{agt}} A'$	performance of external action named P with a possibly empty list of arguments $t_1, ..., t_n$ at artifact j;
2)	$A \overset{msg(i,j,p,c)!}{\longrightarrow_{agt}} A'$	agent i sends message to receiver j with performative p and content c;
3)	$A \overset{msg(i,j,p,c)?}{\longrightarrow_{agt}} A'$	agent j receives message from sender i with performative p and content c;
4)	$A \longrightarrow_{agt} A'$	performance of non-observable internal action

The first transition indicates that an individual agent configuration A changes to A' when it executes an external action. This execution broadcasts an event to the multi-agent level that carries the information about the performed action. This broadcasting is indicated by an exclamation mark. The event is assumed to contain the identifier of the performer, the identifier of the artifact it is fired upon, and the values of the actions parameters. As we shall see later on, this event will be passed to the artifact at the multi-agent level. The second and third transitions involve the sending and receiving of communication messages. Besides changing the agents configuration, the execution of a send action broadcasts an event that is observable at the multi-agent level and which contains the message information, including the sender, receiver, performative and content. The reception of a message is indicated by a question mark. Of course, the reception may also change the state of the agent. Finally, the fourth transition indicates the performance of an internal, non-observable action by an agent. Examples of such internal actions are the updating of the agent's beliefs and the adoption of a plan.

3.4.3 Artifact Level Transitions

An organizational artifact is uniquely identified by its name and is composed of a brute state pertaining to the application specific state, institutional state containing information about role enactment and norm violations, a set of effect specifications by which the outcomes of external actions are determined, as list of events pertaining to the perceived actions fired upon the artifact, and a set of events denoting the actions that are about to be performed and messages to be sent. The notion of artifact configuration is formally defined by the definition below. These notions define the core ingredients of an artifact. In subsequent chapters this definition will be extended to include norms, a richer notion of roles and constructs for changing the norms and organizational structure at runtime.

Definition 3.5 (Artifact Configuration) *An organizational artifact, typically denoted by* O*, is defined as a tuple* $\langle id, \sigma_b, \sigma_i, \mathrm{E}, \epsilon_{in}, \epsilon_{out}\rangle$*, in which:*

- *id is a name uniquely identifying the artifact;*
- *σ_b is a set of ground first-order atoms describing the brute state of the artifact;*
- *σ_i is a set of ground first-order atoms describing the artifact's institutional state;*
- *E is a set of effect specifications;*
- *ϵ_{in} is a list of ground first-order atoms, the events perceived by the artifact;*
- *ϵ_{out} is a list of ground first-order atoms either with predicate name Send or Do, the communication messages to be sent and external actions to be performed.*

We typically write lists as $[\phi_1, ..., \phi_n]$ *and we use '$:$' to denote the concatenation operator. That is,* $[\phi_1, \phi_2] : [\phi_3, ..., \phi_n] = [\phi_1, \phi_2, \phi_3, ..., \phi_n]$.

The first configuration of a computation is determined by the program specifying the artifact. Such a configuration is referred to as initial configuration. An artifact's program only defines the artifact's name, its initial brute state and the effect specifications. Initially, no events are received and no events need to be sent.

Definition 3.6 (Initial Artifact Configuration) *An organizational artifact* $\langle id, \sigma_b, \emptyset, \mathrm{E}, \emptyset, \emptyset\rangle$ *specified by a program such that id is specified by the program's name component,* σ_b *is characterized by the program's facts component and* E *is defined by the program's effect component is called an initial artifact configuration.*

In defining the transition rules explaining the execution of an organizational artifact, we assume an artifact configuration $\langle id, \sigma_b, \sigma_i, \mathrm{E}, \epsilon_{in}, \epsilon_{out}\rangle$. In what follows, the components *id* and E, by which the effects of actions are specified, will be omitted, because these components will not change during computation.

Agents perform actions to interact with the artifact. They may perform an *enact* action to enact a role, a *deact* action for de-enacting a role, and external actions for altering the brute state. Because agents and artifacts are typically distributed, actions are initiated outside the scope of the artifact. The artifact perceives the actions as events which are stored to handle them later on. More precisely, the artifact appends the perceived actions to the end of the list of received events. The perception of actions by the artifact is defined by the following transition rule.

Transition Rule *Let $P(i, id, t_1, ..., t_n)$ be an action where P denotes the name of the action (including the special actions enact and deact), id the identifier of the artifact, i the identifier of the performer, and $t_1, ..., t_n$ the remainder action's arguments. Then the rule for perceiving actions is defined as follows:*

$$\frac{}{\langle \sigma_b, \sigma_i, \epsilon_{in}, \epsilon_{out}\rangle \xrightarrow{P(i,id,t_1,...,t_n)?}_{org} \langle \sigma_b, \sigma_i, \epsilon_{in} : [P(i, t_1, ..., t_n)], \epsilon_{out}\rangle} \quad (rec)$$

Actions that are perceived last are thus inserted as last element of the list. Note that the *id*, pertaining to the identifier the action is performed upon, is left out after storing the action. How the effect of these actions are computed is not defined by this transition rule. Additional transition rules that explain how the artifact configuration changes under the performance of actions are defined presently. These transition rules are defined such that the first element of the list of perceived actions is handled first, i.e. they are processed on a first-come first-served basis. Defining alternative strategies such as priority ordering boils down to redefining transition rule *rec*. This is, however, beyond the scope of this thesis.

The brute facts evolve under the performance of external actions. How the brute state may change by these actions is defined by the effects specification. To determine the effect of an external action is to apply an applicable effect specification. An effect specification (Φ, α, Ψ) is applicable if and only if α unifies with the perceived action for some substitution θ_1, and precondition Φ with θ_1 applied to it can be entailed by the brute state for some substitution θ_2. To apply the effect specification is then to update the brute state with the set of facts that results after consecutively applying θ_1 and θ_2 to postcondition Ψ. Recall that the special atoms starting with predicate symbol `Send` for communicating with other agents and the special atoms starting with predicate symbol `Do` for acting upon other artifacts are not added to the brute state. They are asserted to the set of out events to be handled later on. The process of successfully applying an effect specification for an external action is explained by the following transition rule.

Transition Rule *Let $pred(\phi)$ be a function that evaluates to the predicate name of first-order atom ϕ. Then the rule for processing an external action by applying an effect specification is defined in the following manner:*

$$\frac{(\Phi, \alpha, \Psi) \in \mathrm{E} \quad \mathit{unify}(\alpha, \alpha') = \theta_1 \quad \sigma_b \models \Phi\theta_1\theta_2}{\langle \sigma_b, \sigma_i, [\alpha'] : \epsilon_{in}, \epsilon_{out} \rangle \longrightarrow_{org} \langle \sigma_b', \sigma_i, \epsilon_{in}, \epsilon_{out}' \rangle} \qquad (\mathit{effect})$$

$$\begin{aligned} \text{where } \sigma_b' &= \sigma_b \uplus \{\phi \mid \phi \in \Psi\theta_1\theta_2 \text{ and } pred(\phi) \notin \{Send, Do\}\} \\ \epsilon_{out}' &= \epsilon_{out} : [\phi \mid \phi \in \Psi\theta_1\theta_2 \text{ and } pred(\phi) \in \{Send, Do\}] \end{aligned}$$

The above transition rule pertains to the case in which an effect specification is applicable. However, an action might also fail. For example, because there is no specification for the action at all, or because there is no effect specification of which the precondition is satisfied. In such a case the action is removed from the list of received actions and the agent is notified about the action failure. As the action will not be applied, the brute and institutional state of the artifact remain unchanged. This is defined by the following transition rule. Note that the message informing the agent about the failure is not sent instantaneously, but is stored in the set of out events to be processed later on.

Transition Rule *Let α be an external action performed by an agent identified by i. Then the rule for action failure is defined as follows:*

$$\frac{\nexists (\Phi, \alpha, \Psi) \in \mathrm{E} : (\mathit{unify}(\alpha, \alpha') = \theta_1 \text{ and } \sigma_b \models \Phi\theta_1\theta_2)}{\langle \sigma_b, \sigma_i, [\alpha'] : \epsilon_{in}, \epsilon_{out} \rangle \longrightarrow_{org} \langle \sigma_b, \sigma_i, \epsilon_{in}, \epsilon_{out}' \rangle} \qquad (\mathit{effectf})$$

where $\epsilon_{out}' = \epsilon_{out} : [Send(i, \mathit{inform}, \mathit{failed}(\alpha'))]$

To interact with the artifact agents enact roles. At present it is assumed that any agent i may enact any role defined by the artifact by performing an *enact* action. That is to say, no further conditions are assumed for role enactment. Besides its own identifier and the identifier of the artifact, it provides the identifier r of the role it wishes to enact as argument. Successful enactment is indicated by the assertion of an institutional fact $rea(i, r)$. When the agent no longer wants to play the role r, it may be de-enacted by performing a *deact* action, resulting in the retraction of the $rea(i, r)$ fact. Successful role enactment and de-enactment is specified by the subsequent two transition rules.

Transition Rule *The rule for enacting a role is defined as follows:*

$$\frac{r \in \mathrm{Roles}}{\langle \sigma_b, \sigma_i, [enact(i, r)] : \epsilon_{in}, \epsilon_{out} \rangle \longrightarrow_{org} \langle \sigma_b, \sigma_i', \epsilon_{in}, \epsilon_{out} \rangle} \qquad (\mathit{enact})$$

where $\sigma_i' = \sigma_i \cup \{rea(i, r)\}$

Transition Rule *The rule for de-enacting a role is defined as follows:*

$$\frac{\sigma_i \models rea(i, r)}{\langle \sigma_b, \sigma_i, [deact(i, r)] : \epsilon_{in}, \epsilon_{out} \rangle \longrightarrow_{org} \langle \sigma_b, \sigma_i', \epsilon_{in}, \epsilon_{out} \rangle} \quad (deact)$$

where $\sigma_i' = \sigma_i \setminus \{rea(i, r)\}$

Just like the performance of an external action, the (de-)enactment of a role might fail. For one reason because the role is not known to the artifact. For another reason because the agent tries to de-enact a role it is currently not playing. Similar to the failure of external actions, these actions do not change the artifact's brute and institutional state, and the agent is informed about the failure. Erroneous (de-)enactment is specified by the following two transition rules.

Transition Rule *The rule for failure of enacting a role is defined as follows:*

$$\frac{r \notin \text{Roles}}{\langle \sigma_b, \sigma_i, [enact(i, r)] : \epsilon_{in}, \epsilon_{out} \rangle \longrightarrow_{org} \langle \sigma_b, \sigma_i, \epsilon_{in}, \epsilon_{out}' \rangle} \quad (enactf)$$

where $\epsilon_{out}' = \epsilon_{out} : [Send(i, inform, failed(enact(i, id, r)))]$

Transition Rule *The rule for failure of de-enacting a role is defined as follows:*

$$\frac{\sigma_i \not\models rea(i, r)}{\langle \sigma_b, \sigma_i, [deact(i, r)] : \epsilon_{in}, \epsilon_{out} \rangle \longrightarrow_{org} \langle \sigma_b, \sigma_i, \epsilon_{in}, \epsilon_{out}' \rangle} \quad (deactf)$$

where $\epsilon_{out}' = \epsilon_{out} : [Send(i, inform, failed(deact(i, id, r)))]$

Application of above transition rules possibly adds events to the list of out events, i.e. messages informing an agent about the failure of an action and the side effects the successful execution of an external action may have. Messages that need to be sent and external actions that need to be performed are propagated to the multi-agent level such that their addressee can perceive and transact them. Transition rule *send* defines how messages are sent. Rule *act* defines how external actions are performed by an organizational artifact.

Transition Rule *The rule for sending messages is defined as follows:*

$$\frac{}{\langle \sigma_b, \sigma_i, \epsilon_{in}, [Send(r, p, c)] : \epsilon_{out} \rangle \xrightarrow{msg(id, r, p, c)!}_{org} \langle \sigma_b, \sigma_i, \epsilon_{in}, \epsilon_{out} \rangle} \quad (send)$$

Transition Rule *The rule for performing actions is defined as follows:*

$$\frac{}{\langle \sigma_b, \sigma_i, \epsilon_{in}, [Do(env, P(t_1, ..., t_n))] : \epsilon_{out} \rangle \xrightarrow{P(id, env, t_1, ..., t_n)!}_{org} \langle \sigma_b, \sigma_i, \epsilon_{in}, \epsilon_{out} \rangle} \quad (act)$$

3.4.4 Multi-Agent Level Transitions

Hitherto we defined that agents can perform actions to interact with an organizational artifact, can receive messages sent by an artifact, and we defined how an organizational artifact responds to actions it perceives. Having defined these transition rules we are now able to define the transition system that brings all this together and specifies the execution of a multi-agent system. A multi-agent system is composed of a set of agents that interact with organizational artifacts in pursuing their goals as defined by the following definition.

Definition 3.7 (Multi-Agent Configuration) *A multi-agent configuration is a tuple $\langle \mathbb{A}, \mathbb{O} \rangle$ in which:*

- $\mathbb{A}$ *is a set of heterogeneous agents;*
- $\mathbb{O}$ *is a set of organizational artifacts.*

In defining the multi-agent level transition rules we assume a multi-agent system $\langle \mathbb{A}, \mathbb{O} \rangle$. We start with the rule that defines the communication between organizational artifact and agent. Recall that an artifact may send messages as a result of applying an effect specification and to inform an agent about the failure of executing an action.

Transition Rule *Let $A \in \mathbb{A}$ be an agent identified by j and let $O \in \mathbb{O}$ be an organizational artifact. Then the rule for message synchronization is defined as follows:*

$$\frac{O \overset{msg(i,j,p,c)!}{\longrightarrow_{org}} O' \quad A \overset{msg(i,j,p,c)?}{\longrightarrow_{agt}} A'}{\langle \mathbb{A}, \mathbb{O} \rangle \longrightarrow_{mas} \langle (\mathbb{A} \setminus \{A\}) \cup \{A'\}, (\mathbb{O} \setminus \{O\}) \cup \{O'\} \rangle} \qquad (oamsg)$$

Note that in the above transition rule we do not explicitly assume that artifact O is identified by i. We do so, because in our extension in chapter 5 also roles that are part of the organization may send messages to their players. In that case, the sender of the message will be the role, instead of the artifact.

Although it is assumed that agents interact via an organizational artifact, there is nothing that prevents agents from communicating directly with each other without an artifact knowing about it. Communication between agents is defined by the following transition rule.

Transition Rule *Let $A_1, A_2 \in \mathbb{A}$ be agents identified by respectively i and j. Then the rule for message synchronization is defined as follows:*

$$\frac{A_1 \overset{msg(i,j,p,c)!}{\longrightarrow_{org}} A_1' \quad A_2 \overset{msg(i,j,p,c)?}{\longrightarrow_{agt}} A_2'}{\langle \mathbb{A}, \mathbb{O} \rangle \longrightarrow_{mas} \langle (\mathbb{A} \setminus \{A_1, A_2\}) \cup \{A_1', A_2'\}, \mathbb{O} \rangle} \qquad (aomsg)$$

The following rules define the performance of an observable action by agent or organization that is observed by the organizational artifact the action is intended for. Recall that the perception of actions by the organization is defined by transition rule *rec*.

Transition Rule *Let $A \in \mathbb{A}$ be an agent identified by i and let $O \in \mathbb{O}$ be an organizational artifact identified by id. Then the rule for action synchronization is defined as follows:*

$$\frac{A \xrightarrow{P(i,id,t_1,\ldots,t_n)!}_{agt} A' \quad O \xrightarrow{P(i,id,t_1,\ldots,t_n)?}_{org} O'}{\langle \mathbb{A}, \mathbb{O} \rangle \longrightarrow_{mas} \langle (\mathbb{A} \setminus \{A\}) \cup \{A'\}, (\mathbb{O} \setminus \{O\}) \cup \{O'\} \rangle} \quad (aoact)$$

Transition Rule *Let $O_1, O_2 \in \mathbb{O}$ be organizational artifacts identified by id_1 and id_2 respectively. Then the rule for action synchronization between organizations is defined as follows:*

$$\frac{O_1 \xrightarrow{P(id_1,id_2,t_1,\ldots,t_n)!}_{org} O_1' \quad O_2 \xrightarrow{P(id_1,id_2,t_1,\ldots,t_n)?}_{org} O_2'}{\langle \mathbb{A}, \mathbb{O} \rangle \longrightarrow_{mas} \langle \mathbb{A}, (\mathbb{O} \setminus \{O_1, O_2\}) \cup \{O_1', O_2'\} \rangle} \quad (ooact)$$

The above rules define interaction between artifact and agents that, consequently, change the state of these two components. Both artifact and agent can make internal transitions that only affect their own state. The last two transition rules define the case in which agent and artifact make an internal transition.

Transition Rule *Let $A \in \mathbb{A}$ be an agent. Then the rule for a silent agent transition at the multi-agent level is defined as follows:*

$$\frac{A \longrightarrow_{agt} A'}{\langle \mathbb{A}, \mathbb{O} \rangle \longrightarrow_{mas} \langle (\mathbb{A} \setminus \{A\}) \cup \{A'\}, \mathbb{O}' \rangle} \quad (asilent)$$

Transition Rule *Let $O \in \mathbb{O}$ be an organizational artifact. Then the rule for a silent artifact transition at the multi-agent level is defined as follows:*

$$\frac{O \longrightarrow_{org} O'}{\langle \mathbb{A}, \mathbb{O} \rangle \longrightarrow_{mas} \langle \mathbb{A}, (\mathbb{O} \setminus \{O\}) \cup \{O'\} \rangle} \quad (osilent)$$

3.4.5 Organizational Coordination Cycle

Hitherto, we defined how the constructs of an organizational artifact are executed, but we did not impose any order in which the transition rules are applied. In our approach, such an ordering is imposed by the artifact's coordination cycle. Different strategies for imposing an ordering on the transition rules are possible. We might, for example, decide to receive actions only when no outgoing messages or actions are on stack. Of course, with the small number of tasks our present notion of organizational artifact needs to perform, not many sensible orderings exist. However, as we shall see in the next chapter, when our artifacts also need to enforce norms and apply sanctions, different strategies might be applied on the basis of the problem domain. Questions become relevant such as whether we should always sanction every infringement of the norms, and if we should monitor the compliance of *all* norms before applying sanctions or should monitor the compliance of *one* norm and immediately apply sanctions for a possible violation of that norm. Even more questions arise when we consider constructs for changing the norms at run-time. These questions suggest that multiple strategies for implementing the artifact's coordination cycle exist.

Indeed, in (Aştefănoaei et al., 2009a) the same issue has been discussed by Astefanoaei *et al.* for our counts-as based programming language we presented in (Dastani et al., 2008, 2009). Based on the observation that no single best strategy for applying the rules exists, they propose a meta-level language (based on the language of (Eker et al., 2007)) to define strategies to implement different coordination cycles without affecting the semantics of the programming constructs. This way a clear separation is achieved between the executions at the object level describing the meaning of the programming constructs and the meta-level describing how the artifact may evolve in its computation.

Based on similar ideas as the strategy language of Astefanoaei *et al.* we introduce a meta-level strategy language that describes how the coordination cycle gets executed. Here we introduce a simple language that allows us to describe the execution strategy for the coordination cycle. It should be emphasized that this language is not intended to be part of the programming language (although it might be). Our primary aim is to be able to express different strategies and show the implications different strategies might have for the overall behavior of the organizational artifact. For this aim we borrow the language presented in (Aştefănoaei et al., 2009a). The idea is that we can apply transition rules on a configuration γ. We write $l@\gamma$ to mean that transition rule labeled l (e.g. transition rule *rec* or *effect*) is applied on configuration γ. This expression evaluates to the (possibly empty) set of all configurations applying this rule might result in. The simplest expressions that can be defined in this coordination language are $idle@\gamma$ evaluating to $\{\gamma\}$ and $fail@\gamma$ evaluating to the empty set. Coordination expressions can be concatenated by the operator ;, repeated by the operator $+$ (repeat one or more times) and $*$ (repeat zero or more times) and we can even write conditional expression of the form $E \ ? \ E' : E''$ to mean that if expression E

does not result in the empty set we apply expression E' on its outcome, otherwise we apply expression E'' on the initial configuration. All this is formally defined below.

Definition 3.8 (Coordination Expression Execution) *(Aştefănoaei et al., 2009a) Given coordination expression E and configuration γ, we define the semantics of coordination expressions as follows:*

- $l@\gamma = \{\gamma' \mid \gamma \rightarrow \gamma'$ is derivable by applying transition rule labeled $l\}$
- $idle@\gamma = \{\gamma\}$
- $fail@\gamma = \emptyset$
- $E@\Gamma = \bigcup \gamma \in \Gamma : E@\gamma$
- $(E; E')@\Gamma = E'@(E@\Gamma)$
- $(E \mid E')@\Gamma = E'@\Gamma \cup E@\Gamma$
- $E^{+}@\Gamma = \bigcup i \geq 1 : E^{i}@\Gamma \;$ s.t. $\; E^{1} = E \;$ and $E^{n} = E^{n-1}; E$
- $E^{*}@\Gamma = idle@\Gamma \mid E^{+}@\Gamma$
- $(E \;?\; E' : E'')@\Gamma = \begin{cases} E'@(E@\Gamma) & \text{iff } E@\Gamma \neq \emptyset \\ E''@\Gamma & \text{iff } E@\Gamma = \emptyset \end{cases}$

Note that according to the first bullet applying a rule on a configuration gives us a set of possible outcomes, while one might expect that applying a transition rule gives us only one. However, the application of a rule may have multiple outcomes. Consider, for example, transition rule *effect*. Because this rule does not deterministically pick one applicable effect rule and more than one effect rule might be applicable, this rule can have more than one possible outcome. Building on these expressions we define some auxiliary expressions, namely:

- $not(E) = E \;?\; fail : idle$, i.e. reverse the outcome of E;
- $try(E) = E \;?\; idle : idle$, i.e. apply E only when applicable, and otherwise, do not change the final outcome;
- $test(E) = not(E) \;?\; fail : idle$, i.e. test whether E can be applied without applying it on the final outcome; and
- $E! = E^{*}; not(E)$, i.e. recursively apply E until it is no longer applicable.

With this strategy language we can now specify the coordination cycle defining the order in which the transition rules are applied. In other words, we can define the steps of handling role enactment and processing an external action as depicted in figure 3.4 and 3.5. Consider, for example, the following strategy for applying the effect of an action:

$$rec; (effect|effectf|enact|enactf|deact|deactf) \qquad \text{(action)}$$

which first applies transition rule *rec* for receiving an action. Then, it applies either one of the rules *effect*, *effectf*, *enact*, *enactf*, *deact* or *deactf* for handling the action. Note that these rules are all mutually exclusive, meaning that at most one of the rules is applicable, i.e. will evaluate to a non-empty result. It is important to note that when no action can be received, i.e. the rule *rec* is not applicable, the whole expression evaluates to the empty set. Also note that this expression may evaluate to a set of configurations. Suppose, for instance, there is more than one effect rule applicable. The final outcome of applying the expression *effect* will then contain one configuration for each applicable effect rule. Furthermore, this strategy merely determines the effect of an action. It does not send all outgoing messages that are added to the list of outgoing events, nor does it perform any external actions. To handle all outgoing events we define the following strategy:

$$(send|act)! \qquad \text{(out)}$$

that iteratively applies rule *send*, for sending out all messages, and *act*, for performing all external actions, until no longer applicable.

The two strategies defined above can now be combined in one strategy defining one step of the coordination cycle (we refer to a strategy expression by its name): action; out. This expression gives us all the artifact configurations that may result after applying the transition rules as described by the expression. That is to say, it evaluates to all the possible configurations after executing one step of the coordination cycle. Throughout this thesis we are often interested in one possible outcome of executing the coordination cycle one or more times, and all the intermediate configurations that were used in reaching it. For this purpose we define the concept of a coordinated trace.

Definition 3.9 (Coordinated Trace) *We define the function $C^n_E(\mathrm{O})$ that evaluates to a possible execution trace that results after iteratively applying coordination strategy E n times on organizational configuration O in the following manner:*

$$C^n_E(\mathrm{O}_0) \quad = \quad \mathrm{O}_0\mathrm{O}_1 \cdots \mathrm{O}_n \text{ s.t. for } 0 \leq j < n \;:\; \mathrm{O}_{j+1} = \mathrm{O} \text{ for some } \mathrm{O} \in E@\{O_j\}$$

The coordinated trace, which C evaluates to, allows us to refer to executing the coordination cycle one or more times, abstracting away from the micro-level transitions that underly these executions. Note that C is undefined for the case there is an empty outcome of executing (one of the intermediate steps of) the coordination strategy.

3.5 Discussion

We introduced organizational artifacts for implementing the environmental part of a multi-agent system. Revisiting the responsibilities described in chapter 2 that are assigned to the environment by Weyns *et al.* (2007), organizational artifacts fulfill these responsibilities in the following manner:

Structuring. A multi-agent system is physically structured by a distribution of agents and the organizational artifacts they interact with. Allowing interactions between artifacts facilitates to break up a system in a set of coherent, loosely coupled interacting artifacts, as shown by the conference management system example. This promotes a further distribution of the system. Moreover, organizational artifacts may be deployed for structuring the communication between agents. An agent may use an artifact for communicating with other entities, being oblivious about the details of whom is being communicated with. The view on roles (to be extended in subsequent chapters) accounts for a social structuring of the agent's interactions.

Resource and service embedding. An organizational artifact embeds a domain-specific state by means of brute facts. Functionality the agents may exploit to reach their objectives – i.e. giving them new powers (Castelfranchi, 2003) – is provided by the set of actions that can be used to manipulate the artifact's brute state and send communication messages. The artifact's action specification shields the agents from the low-level implementation details. That is to say, an agent only needs to perform actions to alter the brute state without needing to be concerned with how these brute facts change.

Maintaining dynamics. Our notion of organizational artifact does not directly maintain its own dynamics. However, task-oriented processes may be deployed that act upon an artifact and in this way implement the environment's dynamics. These task-oriented processes are not agent-oriented and deployed separately from the agents.

Local observability. Actions may result in messages being sent informing agents about the state of the organizational artifact. This allows agents to inspect the internals of the artifact. The precondition of an action effect specification can be used for granting permissions to agents to perform these actions. This is typically done on the basis of roles that are played by the agents. The sphere of visibility of an agent is thus determined by the actions it may perform.

Local accessibility. Similar to inspecting, agents may alter the brute state of the artifact by means of performing actions. The sphere of influence of an agent on an artifact's internal state is determined by the actions it may perform.

Regulating. An organizational artifact may be equipped with norms (discussed in the next chapter) to regulate the agents' interactions. These norms can then be used to further specify who the agent may interact with and which actions are (il)legal, thereby restricting the agents' sphere of influence.

An important class of multi-agent systems is the class of systems that are characterized by their openness. In chapter 2 we discussed four types of openness as identified by Davidsson (2001), namely open systems, semi-open systems, closed systems and semi-closed systems (also see chapter 2). Considering the artifact and the agents that directly interact with it as the systems' boundary, our approach in which a basic organizational artifact is used (without norms and positions), can already account for all four types of openness. By not putting any restrictions on which agents may start interacting with the artifact a system that pertains to an open system can be built. Note that because any agent may enact a role, the conference management system can be characterized as an open system. A closed system can be built by specifying the actions in such a way (i.e. by explicitly referring to the identity of each agent that may perform an action in the precondition of the effect specifications) that the collection of agents that can interact with it is fixed. By allowing such a fixed set of agents to interact with arbitrary external agents a semi-closed system is reached. Finally, a semi-open system can be engineered by assigning gatekeeper agents with the responsibility to decide whether an external agent is entitled to join. If an external agent is entitled to join the gatekeeper registers this agent to the artifact which will change the brute state in such a manner that now actions are enabled for this newly joined agent. Note that the way in which semi-open and semi-closed systems are built with basic artifacts, the burden of the task of deciding who may interact with the organization and the task of acting as an interface between the artifact and the agents is on the agents themselves. In subsequent chapters we will introduce more powerful mechanisms, i.e. norms and positions, alleviating the agents from this burden.

Before ending this chapter it is important to point out the relation with the work on the agents and artifacts approach to engineering multi-agent systems as introduced by Ricci *et al.* (2006; 2007). Based on the same idea as the A&A approach the agents' environment is enriched with artifacts providing non-agent functionality the agents may exploit in pursuing their goals. Differently from their approach, however, our approach does not provide a usage interface defining the operations which agents can invoke and operating instructions describing how to use these operations to fruitfully exploit the artifact's functionality. This is left for future research. Another major difference is that the A&A approach is a meta-level engineering approach, whereas we describe a concrete programming language, by which artifacts can be programmed, in terms of a formal syntax and operational semantics.

3.6 Conclusion

In this chapter we introduced the notion of organizational artifact as a tool that agents can exploit in achieving their objectives. An organizational artifact is built separately from the agents and deployed in the environment of a multi-agent system. We discussed the essential syntactical constructs involved in programming an organizational artifact and we demonstrated how to use them by means of a conference management system example. We endowed the syntax with an operational semantics. Finally, we discussed to what extent our notion of organizational artifact fulfills the responsibilities an environment is supposed to fulfill according to Weyns *et al.* (2007), and we have discussed how artifacts may be used for building different types of open systems. In subsequent chapters the syntax and semantics of the basic organization-oriented programming language will be extended to include more organization-oriented constructs such as roles and norms, as well as constructs for changing the norms.

Chapter 4

Programming Regulative Norms

In the previous chapter we introduced the notion of organizational artifact as a useful tool that resides in the environment of the agents. In the class of systems that can be characterized by a degree of openness in which agents that are not a priori known dynamically start interacting with an organizational artifact (cf. (Davidsson, 2001)), also little can be assumed about the behavior these agents will exhibit. When unknown agents start their interactions dynamically and little can be assumed about the features of these interactions, there is a strong need for mechanisms to regulate their behavior in order to guarantee the global objectives of the artifact (and the system as a whole) to be achieved or maintained. Agents may, for example, (deliberately or accidentally) exhibit behavior that obstructs the system's global goals. The use of an explicit representation of normative concepts such as obligations, prohibitions and sanctions has been widely promoted as a suitable tool for such regulation. Indeed, much literature can be found on the formal specification and verification of norms ((Meyer and Wieringa, 1993), (Prakken and Sergot, 1996), (Dignum et al., 2004), (Sergot and Craven, 2006), (Boella et al., 2008) and also our own work (Dastani et al., 2008) are just but a few examples), and literature on the practical issues associated to operationalizing norms for their use in computational (multi-agent) systems e.g. (Esteva et al., 2004; Garcia-Camino et al., 2005; Silva, 2008; Minsky and Ungureanu, 2000; Dastani et al., 2009) (more examples can be found in chapter 2).

Despite these developments we observe a gap between research on the construction of normative frameworks and multi-agent programming languages. We conjecture that the following two issues underly the root cause of this gap. First, all of the above mentioned implementations (except for (Dastani et al., 2009;

Silva, 2008)) are primarily targeted at “procedural” norms specifying which actions agents ought or ought not perform and disregard the issue of expressing “declarative” norms related to a description of a desired state of affairs that should be brought about in an environment the agents interact with. Norms in existing frameworks thus typically take on the form of “ought-to-do” statements pertaining to actions rather than “ought-to-be” statements pertaining to the declarative description of a state. We argue that expressing declarative norms is also important, because:

1. If we can relate to actions only, writing norms to ensure a certain state is achieved might become a rather tedious task, especially when establishing it involves a multitude of actions and the endeavor of multiple agents. It is, for example, difficult to express a norm for a conference management system that at least two reviews should be received for each paper;

2. By only stating the actions the agents should perform to reach a desired situation we risk to limit their autonomy, because we leave them no choice in deciding how to reach it. For example, by telling an agent to take the train to a conference we leave it with less options than by just telling it to be at the conference;

3. Related to the previous point, expressing declarative norms accords more with the concept of declarative goal as often used for modeling agents, viz. a description of a desirable situation. This conformity facilitates the internalization of norms by agents for reasoning with them (see for example (Meneguzzi and Luck, 2009)). In the above example, the agent probably has multiple plans to achieve the goal of being at the conference, such that this obligation can be more easily internalized and acted upon.

Second, current work on normative frameworks for regulating the behavior of agents is not endowed with an operational semantics (Plotkin, 1981) that allows for a direct implementation of an interpreter. An operational semantics allows us to evaluate a framework by formally studying the key properties it exhibits and it relates it to some of the results established by the field of deontic logic. Many agent-oriented programming languages are already investigated by means of an operational semantics and also doing this for the normative concepts contributes to the longer-term goal of studying the properties of a multi-agent system regulated by norms.

In this chapter we aim to narrow this gap by extending the basic idea of an organizational artifact to include normative concepts for regulating the agents' interactions with the aim of preventing the artifact from ending up in sub-ideal states. More concretely, the main contributions of this chapter are:

- We describe the overall architecture of a multi-agent system regulated by normative components in section 4.1. In particular, we explain our norm enforcement mechanism in the context of an organizational artifact as presented in previous chapter and motivate our choice of the involved normative constructs.

- We introduce the syntax and intuitive semantics by which norms can be programmed in section 4.2. Our norms take on the form of conditional obligations and prohibitions with a deadline. Their representation and semantics is inspired by norms as presented by Boella *et al.* (2008) who only consider obligations. Contrasting existing work on computational norms (except for (Dastani et al., 2009; Silva, 2008)) that is primarily targeted at “procedural” norms specifying which actions agents ought or ought not perform, we consider “declarative” norms related to a description of a desired state of affairs that should be brought about in some environment the agents interact with.

- To motivate agents to abide by the norms we introduce sanctioning rules which are taken from our earlier work on counts-as rules (Dastani et al., 2008, 2009; Tinnemeier et al., 2009a) (also in section 4.2). The solution we develop allows both for regimentation and enforcement of the norms (cf. (Aldewereld, 2007; Castelfranchi, 2000; Grossi, 2007)). We demonstrate how these norms and sanctioning rules can be used to regulate the agents' behavior and to prevent an organizational artifact from ending up in unwanted situations by means of our conference management system example (section 4.2).

- We endow the syntactical constructs with a structured operational semantics (Plotkin, 1981) in section 4.3. This enables us to study the normative concepts in a rigorous manner and an operational semantics is already close to the implementation of an interpreter without committing to a particular programming language. Moreover, it reduces the gap between research on agent programming languages (that are often endowed with an operational semantics) and research on computational, normative frameworks (that are rarely studied by an operational semantics).

- We demonstrate that different strategies exist for enforcing the norms in section 4.4. This has not been done before in respect of a computational normative framework (except for (Aştefănoaei et al., 2009a)). Moreover, in this section we show some of the key properties our norms exhibit, relating them to some of the properties that have been long studied by the field of deontic logic and normative multi-agent systems. This relation is generally overlooked by existing work on computational frameworks for norm enforcement.

4.1 Organizational Artifacts with Norms

As previously explained in chapter 3, we conceive a multi-agent system as consisting of a collection of heterogeneous agents and a collection of artifacts. The agents exploit the functionality provided by organizational artifacts to achieve their goals. An organizational artifact implements the non-autonomous functionalities that are better implemented by non-agent concepts. They encapsulate a domain specific state and function, which is modeled by a set of *brute facts*. The agents perform actions that change the brute state to interact with the artifact and exploit its functionality. An overview of the internals of an organizational artifact is shown in figure 4.1. In this chapter we will focus on the normative dimension of the organizational artifact.

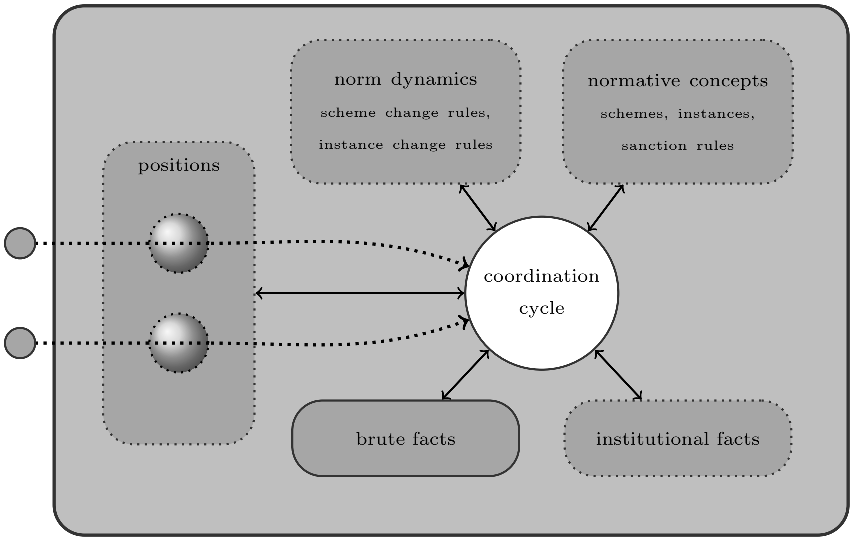


Figure 4.1: A (simplified) conceptual representation of the internals of an organizational artifact. Optional concepts (e.g. norms, positions) are outlined with a dashed border. Solid arrows denote the reading and modification of the concepts as explained in more detail below. Dotted arrows denote actions and messages between agent and artifact. A brief description of all the concepts can be found in chapter 3.

When unknown agents of which little can be assumed about their behavior interact with the artifact, it is essential to coordinate their behavior and to guide them in interacting with it in a meaningful way. What is considered to be desired behavior is described by the *norm schemes*, a set of conditional obligations and prohibitions. The norm schemes define under which conditions obligations and prohibitions should be created. For example, if a reviewer is assigned a paper to review, then an obligation to have uploaded a review for that paper is created. The condition of a norm scheme relates to the brute and institutional state of the artifact and whenever its condition is satisfied the artifact instantiates the obligation or prohibition belonging to it, hence the name *norm instance.* The process of triggering is summarized in figure 4.2.

The norm instances that are created out of the norm schemes are thus a set of active (unconditional) obligations and prohibitions. They specify which brute and institutional states should (not) be achieved. Seeing to it that the norms and their instances are about the brute and institutional state an artifact encapsulates and maintains, we consider it the artifact's responsibility to detect when obligations and prohibitions should be created and detect violations and fulfillments. We refer to this process by the name *monitoring* as shown in figure 4.3. Assigning this responsibility to third-party entities or even to agents, would break the principle of data hiding. Moreover, delegating this responsibility to another entity with its own thread of control introduces nasty synchronization issues. In particular, putting the burden of monitoring on designated 'enforcement' agents, requires advanced reasoning capabilities of those agents. In chapter 2 we showed that such capabilities are beyond the reasoning capabilities agents typically have, and additional mechanisms would be needed to verify if the monitoring agents carry out their task satisfactorily. To conclude, assigning the responsibility of monitoring to external entities does not buy us anything, it would be merely shifting the problem. We do note, however, that this is differently from how it is done in the real world where, for example, policemen are used to detect traffic violations. The point we make here is that if the system itself can detect violations, monitoring should be done by the system. Indeed, the same trend can be observed in real life where the tasks of policemen are often replaced by automatic devices, e.g. speeding cameras.

Usually, a norm instance is accompanied by a deadline (cf. (Dignum et al., 2004; Boella et al., 2008)). For example, an obligation to have reviewed a paper should be fulfilled before the notification phase starts. Intuitively, an obligation specifies which brute state should be established before a particular moment, whereas a prohibition specifies which brute state should be avoided until the deadline. Resembling the specification of deadlines as propositional formulae instead of time (Boella et al., 2008), in our approach, deadlines are expressed as a situation that can be entailed by the brute state. This gives us more freedom in specifying different notions of deadlines and allows us, for example, to relate to a situation that is established by the actions of other agents without the need to know if and when it will be established.

A special type of norm that has also been discussed in chapter 2, is the one that specifies which obligations or prohibitions should be instantiated in the sub-ideal situation in which some other norm instance is violated. Contrary-to-duty norms (Prakken and Sergot, 1996) are a special case of this. For example, a reviewer that did not abide by its obligation to have reviewed a paper obliges the program chair to remind him/her of this task. Recall that information about which norm

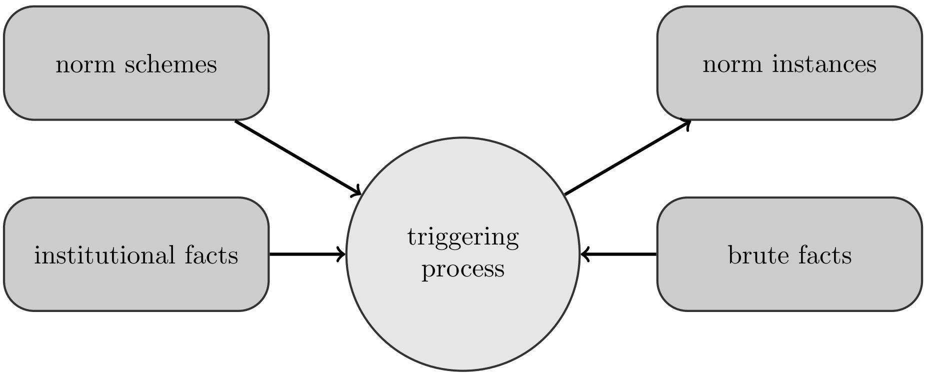


Figure 4.2: Triggering amounts to checking if the condition of a norm scheme can be entailed from the current brute and institutional state. If so, an obligation or prohibition is created and added to the set of norm instances.

instances are violated is stored by the *institutional facts* (as taken from (Searle, 1995)). To also allow for the specification of these types of norms we let the condition of norm schemes relate to institutional facts. We call a norm that is triggered by some condition of the brute state a primary norm, and a norm that rises due to the violation of another norm a secondary one. This resembles the notion of interlocking of norms as proposed in (Lopez y Lopez et al., 2006).

Usually, a norm instance is directed at someone (Lopez y Lopez et al., 2006), but because the artifact's interactants are not known beforehand, in expressing the norm schemes we rather refer to the roles that are played by agents instead of to the agents directly. For the present work we adopt the same simple representation of roles and their enactment as we did in the previous chapter. A role is a label r that identifies it by a unique name. We model the institutional fact that an agent identified by a has enacted role r by a proposition $rea(a, r)$ as stored by the institutional facts. In subsequent chapters we will change this representation of roles to include a richer account.

To motivate agents to abide by the norms, there are often sanctions associated to the violation or fulfillment of a norm (cf. (Vázquez-Salceda et al., 2008; Lopez y Lopez et al., 2006)). These sanctions include punishments for an agent who infringes the norms and rewards for an agent who obeys the norms. For example, an author that violates a prohibition to upload a paper of more than 15 pages is punished by an instant rejection of the paper by removing it from the database. A question that rises is "who should be responsible for applying these sanctions?" We already argued that the organizational artifact is responsible for activating obligations and prohibitions based on their underlying norms, and detecting violations and fulfillments of these active obligations and prohibitions. Based on a similar reasoning we could conclude that also sanctioning should be the artifact's

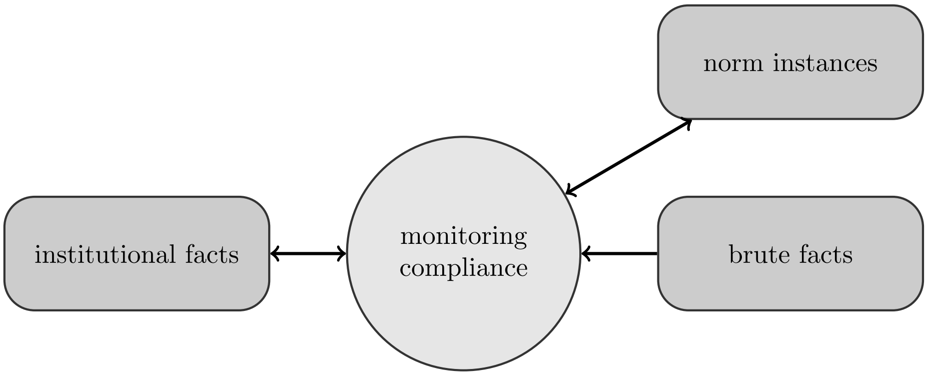


Figure 4.3: Monitoring compliance of an obligation amounts to checking if the desired situation (of the brute and institutional state) can be entailed before the deadline. Violated and fulfilled obligations are removed from the set of norm instances. Monitoring obedience of a prohibition boils down to checking if the undesired situation can be entailed before the deadline. Prohibitions are removed when their deadline can be entailed. Information about violation and fulfillment is added to the institutional facts.

task. Additionally, sanctioning might require to modify the brute facts as our example of removing a paper suggests. However, sanctioning does not require advanced reasoning capabilities of the agents; they only need to know that someone has complied to or has infringed the rules without needing detailed knowledge about how the norms are expressed. This pleads for a mechanism that at the very least allows the artifact to notify agents about (non)compliance and leaving the decision and act of sanctioning to them, but at the very same time allows for the artifact to apply sanctions automatically.

Note that not every norm has a sanction associated to it. One might, for example, decide not to punish the infringement of a norm, but only reward compliance (or vice versa). Based on this observation and the fact that sanctioning is a different concern than norm activation and verifying compliance, we argue that the specification of sanctions and the process of applying them should be sensibly separated out as a different component. In our approach, sanctions are specified by so-called sanctioning rules taken from (Dastani et al., 2008, 2009; Tinnemeier et al., 2009a). They take on the form of a kind of reversed counts-as rules that specify which brute facts should be accommodated to the brute state of the artifact as a consequence of a violation or fulfillment of some obligation or prohibition. To enable the artifact to inform agents and other artifacts about (non)compliance we allow these reverse counts-as rules to have the same side effects as the performance of actions, namely messages to be sent or actions to be performed upon other artifacts.

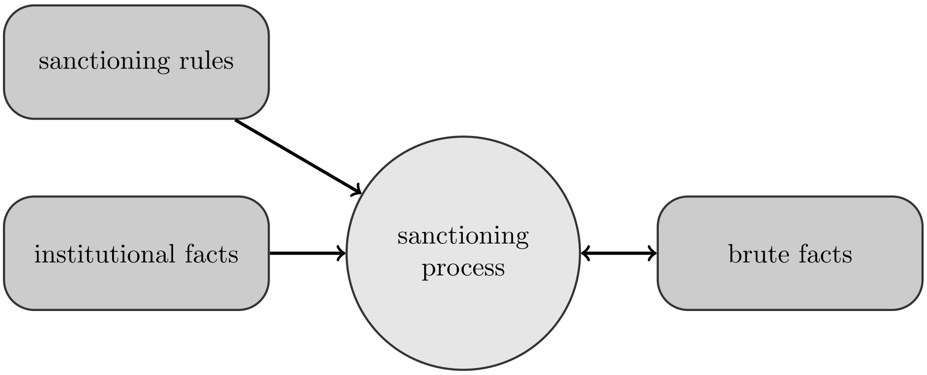


Figure 4.4: Sanctioning amounts to accommodating the brute facts that appear in the consequents of applicable sanctioning rule to the brute state. A sanctioning rule is applicable when its antecedent is satisfied by the brute and institutional state.

4.2 Programming the Normative Dimension

In this section we describe the syntax of the programming constructs by which the normative component of an organizational artifact can be defined. We use the conference management system example to show how norms can be programmed and to explain their intuitive semantics. Recall that to program a basic normative artifact is to specify the roles agents can play, a set of facts specifying the initial brute state and a set of effects specifying how the brute state evolves under the performance of actions. The grammar of these constructs can be found in table 3.1 of chapter 3. All these constructs were explained in detail in that chapter and will not be repeated here. We do, however, repeat some of the elementary syntactical constructs in table 4.1 that were originally listed in table 3.1 and extend them, because they are needed for defining the syntax by which norms can be programmed. To extend our notion of organizational artifact to include norms, we extend the previously defined clause ⟨artifact⟩ as defined in figure 3.6 of chapter 3 in the following manner:

$$\begin{array}{lcl} \langle\text{artifact}\rangle & = & \texttt{"Name:"}\ \langle\text{id}\rangle\ [\ \langle\text{roles}\rangle\ ]\ [\ \langle\text{enacts}\rangle\ ]\ [\ \langle\text{facts}\rangle\ ]\ [\ \langle\text{effects}\rangle\ ] \\ & & [\ \langle\text{norms}\rangle\ ]\ [\ \langle\text{sanctions}\rangle\ ]; \end{array}$$

The grammar of norms and sanctions is shown in figure 4.5. In what follows, we explain the intuitive semantics of the syntactical constructs by our conference management system example.

Norm schemes and Norm Instances

The conference management system has some behavioral expectations about its participants. We expect, for example, reviewers to review their assigned papers

⟨label⟩	a first-order atom with constants and variables used to uniquely identify norms
⟨b−atom⟩	a first-order atom denoting a brute fact. The special facts starting with predicate symbol *viol*, *obey* and *rea* (their meaning to be explained later on) are excluded from the set of brute facts.
⟨i − atom⟩	a first-order atom of the form $viol(\phi_l)$ or $obey(\phi_l)$ with ϕ_l a ⟨label⟩ identifying a norm. The first denotes a violation of norm labeled ϕ_l whereas the latter denotes the obedience to the norm.
⟨r − atom⟩	a first-order atom of the form $rea(i, r)$ in which r denotes a role and i the agent playing it; "rea" is short for role enacting agent.
⟨send⟩	a first-order atom of the form $Send(r, p, c)$ in which r denotes the message's receiver, p its performative and c its content.
⟨do⟩	a first-order atom of the form $Do(id, \alpha)$ in which id denotes the artifact upon which action α is to be performed.

Table 4.1: Elementary syntactical constructs.

```
⟨norms⟩      = "Norms:" ⟨norm⟩ { ⟨norm⟩ };
⟨norm⟩       = ⟨label⟩ ":   <" ⟨cond⟩ "," ⟨OP⟩ "," ⟨ddln⟩ ">";
⟨cond⟩       = ⟨b − lit⟩ | ⟨r − lit⟩ | ⟨i − atom⟩ | ⟨cond⟩ "and" ⟨cond⟩;
⟨OP⟩         = "O(" ⟨assertion⟩ ")" | "F(" ⟨assertion⟩ ")";
⟨ddln⟩       = ⟨assertion⟩;
⟨assertion⟩  = ⟨b − lit⟩ | ⟨r − lit⟩ | ⟨test⟩ "and" ⟨test⟩;
⟨sanctions⟩  = "Sanctions:" ⟨sanction⟩ { ⟨sanction⟩ };
⟨sanction⟩   = ⟨presanc⟩ "=>" "{" ⟨postsanc⟩ "}";
⟨presanc⟩    = ⟨b − lit⟩ | ⟨r − lit⟩ | ⟨i − atom⟩ | ⟨presanc⟩ "and" ⟨presanc⟩;
⟨postsanc⟩   = ⟨b − lit⟩ | ⟨send⟩ | ⟨do⟩ | ⟨postsanc⟩ "," ⟨postsanc⟩;
⟨b − lit⟩    = "true" | ⟨b − atom⟩ | "not" ⟨b − atom⟩;
⟨r − lit⟩    = ⟨r − atom⟩ | "not" ⟨r − atom⟩;
```

Figure 4.5: EBNF grammar of norm schemes.

in time. Such behavioral expectations are are expressed by the norm schemes, that is, conditional obligations and prohibitions. In our framework, we do not consider permissions; if something is not explicitly forbidden, we assume it to be permitted.[1] A conditional obligation is expressed as a labeled tuple of the form $\phi_l : \langle \varphi_c, O(\varphi_x), \varphi_d \rangle$ with the intuitive reading that "if condition φ_c holds then there is an obligation to establish the brute state denoted by φ_x before deadline φ_d". A conditional prohibition is expressed as a labeled tuple $\phi_l : \langle \varphi_c, F(\varphi_x), \varphi_d \rangle$ that can be intuitively read as "if condition φ_c holds then it is forbidden to establish the brute state denoted by φ_x before deadline φ_d." Note that each φ

[1]Here we assume that permission is the dual of prohibition. It should be noted that there are also interpretations in which this is no longer the case, see for example (Hansen et al., 2007) for a discussion.

Code fragment 4.1 Norms and sanctions of reviewing system

```
Norms:
page_size(PId):
  < phase(submission) and abstract(A,PId)
   , F(pages(PId) > 15)
   , phase(review) >

review_due(R):
  < phase(review) and assigned(R,PId)
   , O(review(R,PId))
   , phase(collect) >

minimum_reviews(PId):
  < phase(submission) and paper(PId)
   , O( nr_reviews(PId) >= 3 )
   , phase(collect) >

viol_minimum(C):
  < viol(minimum_reviews(PId)) and rea(C,chair)
   , O(review(C,PId))
   , phase(notification) >

Sanctions:
viol(review_due(R)) => {blacklist(R)}

viol(viol_minimum(C)) => {blacklist(C)}

viol(page_size(PId)) and paper(A,PId) => {false}
```

is a conjunction of brute and institutional literals. By allowing the precondition to contain *rea* atoms, we can associate norms to roles that are played by agents. The parameterized labels ϕ_l the norms are equipped with, is used to uniquely identify them and to keep track of which of them are violated. Remember that we distinguish between primary norms that are triggered because of a particular condition of the brute state and secondary norms that are triggered due to the violation of another norm. The condition of primary norms thus relates to brute facts, whereas the condition of norms at level one and beyond relates to violation and obedience facts as stored in the institutional state. We acknowledge that norm schemes can contain more components as, for example, explained in (Lopez y Lopez et al., 2006; Vázquez-Salceda et al., 2004). Nevertheless, we limit ourselves to the elementary components that are needed to explain the semantics of the enforcement mechanism. Other components such as addressees and beneficiaries and rewards can be incorporated without any technical difficulties, but including them will obfuscate the explanation of the norms' semantics.

The norm schemes of the reviewing system, expressing the desired behavior of the agents playing the role of reviewer, author or chair are listed in code fragment 4.1. The first norm scheme expresses that uploaded papers should not exceed the page limit of 15 pages. Suppose an author, say `jane`, has uploaded an abstract and has been assigned id. `547`. As soon as the chair puts the system in the submission phase the condition is satisfied and the norm scheme is instantiated

Code fragment 4.2 Norms and sanctions of registration system

```
Norms:
author_early_reg(A):
  < author(A) and phase(early) and not registration(A,author,complete)
  , O(registration(A,author,complete))
  , phase(registration) >

Sanctions:

viol(author_early_reg(A)) => {Send(A,request,register)}

fulfill(author_early_reg(A)) => {free_banquet_ticket(A)}
```

into a norm instance pertaining to a prohibition `F(page_size(547) > 15)` stating that `jane`'s paper is not allowed to exceed 15 pages. This prohibition stays into effect during the whole submission phase, i.e. until the review phase starts which is expressed by the deadline `phase(review)`. A violation is detected as soon as `jane` uploads a paper of more than 15 pages. Note that a norm scheme may instantiate multiple norm instances; they are implicitly universally quantified in the widest scope.

To be able to make decisions about which papers to accept and which ones to reject during the collect phase, reviewers are required to have uploaded their reviews before the end of the reviewing phase. This is specified by the second norm scheme, which states that a reviewer is obliged to have uploaded its assigned reviews before the reviews are collected. This obligation becomes active as soon as a reviewer is assigned a paper and stays into effect until it is either fulfilled (the review has been uploaded) or it is violated (the review has not been uploaded before the deadline).

To increase the quality of the reviewing process, the third norm specifies that by the end of the reviewing phase there should be at least three reviews per paper. In the sub-ideal situation this norm is violated, the chair is expected to take charge of reviewing this paper as expressed by the fourth norm scheme. To explain this norm scheme, suppose that when the review phase has ended still only two reviews for paper `547` have been received. Then the obligation `O(nr_reviews(547)>=3)` that emanated from the second norm scheme is violated. This violation is marked by the assertion of an institutional fact `viol(minimum_reviews(547))` to the institutional state. This will trigger the fourth norm scheme, that will instantiate an obligation for the chair to have uploaded a review for paper `547` before the notification phase starts. Note how parameters of labels are used to pass information between norm schemes at different levels. It should be emphasized that the parameters of the labels are formal output parameters that become actual parameters upon instantiation of the norm scheme. By allowing the condition of a norm scheme to contain institutional facts that contain information about norm violations, we can express obligations and prohibitions that are effectuated when certain norms are broken (or abided by).

Remember that in our example participants should first register themselves to the registration system (authors of accepted papers are automatically registered by the reviewing system). Registering alone is not enough, a registration is considered completed if the conference fee has been paid for. Authors of accepted papers are requested to complete their registration in time, preferably already during the early registration. This is expressed by the first norm scheme of code fragment 4.2 specifying that authors should complete their registration during the early registration phase. Early registration is preferred, but not a hard requirement. In fact, as we shall see later on, no punishment is associated to the violation of this obligation.

Sanctioning Rules

To make agents respect the norms, sanctions are often imposed to punish infringements or reward obedience. Sanctions (taken from (Dastani et al., 2008, 2009; Tinnemeier et al., 2009a)) are expressed separately from the norms as a kind of inverted counts-as rules. They associate a sanction with a normative assessment of the artifact's situation as modeled by the brute and institutional facts. More concretely, they are expressed as rules of the form φ => Ψ which can be intuitively read as: "a normative situation (as modeled by the brute and institutional facts) expressed by φ is sanctioned by accommodating brute facts Ψ to the artifact's brute state."

The sanctioning rules of the reviewing system are listed on lines 22-27 of code fragment 4.1. The first sanctioning rule of the reviewing system states that reviewers who fail to have uploaded their reviews in time are blacklisted, i.e. a decrease in reputation. This information could be used in deciding who should be on the programme committee of a future conference. The same sanction is applied to the chairs who do not conform to the obligation to review a paper for which less then three reviews have been uploaded. This is expressed by the third sanctioning rule. Again, note how the parameters of the norm scheme in combination with the violation (and obey) facts are used to pass on information about the actual norm instance that has been violated (or abided by). The third sanctioning rule of the reviewing system is an example of regimentation, cf. (Aldewereld, 2007; Castelfranchi, 2000; Grossi, 2007; Jones and Sergot, 1993). Recall that regimentation boils down to *ruling out* all the actions that will lead to an intolerable state, such that a violation of the norms will never happen. The fact that this state is intolerable is denoted by the special atom `false`. The third sanctioning rule thus says that it is impossible for an author to upload a paper that exceeds the page limit. Even though an author might still try to upload a paper of more than fifteen pages, the effect of this action will not be effectuated by the organizational artifact.

Recall the norm specifying that authors of accepted papers should register for the conference, preferably during early registration. To motivate these authors to register already during early registration, we associate a reward to the fulfillment of the obligation to register before the registration phase starts (i.e., the first norm of the registration system.) The reward that is imposed is a free ticket for the banquet, which is presumably handed out at the registration desk. As a consequence of violating this obligation a reminder to register is sent to the author concerned. Because this norm pertains to a suggestion expressing preferable behavior instead of required behavior, no punishment is involved. Both sanctions are expressed by the first and second sanctioning rule listed on lines 7-11 of code fragment 4.2.

Ideally, an author who fails to register before the registration system closes is punished by automatically rejecting all its formerly accepted papers. Expressing such a norm and associated sanction, however, turns out not to be trivial. Information about the registration of authors is stored in the registration artifact. This suggests that the norm which obliges authors to register should be implemented as part of the registration artifact. This registration artifact, however, does not have information about which papers belong to that author. This information is stored by the reviewing artifact. The underlying problem is that the information that is needed to enforce this rule is spread across different artifacts. In this case we can overcome this problem by (for example) letting the registration artifact inform the reviewing artifact about the registration status of authors. Then, the norm and sanctioning rule can be implemented as part of the reviewing system. However, this introduces the storage of redundant information, which may lead to nasty synchronization issues. Besides, we can also imagine comparable problems that are not so easily fixed by letting artifacts exchange information. Suppose, for example, we would break up our reviewing system in different organizational artifacts each implementing the reviewing functionalities of a conference sub-track. How to express a norm scheme specifying that a reviewer can only be on the program committee for one sub-track or a norm scheme that specifies that not more than a certain amount of papers may be accepted for the whole conference is unclear. We believe the problem to be a fundamental one; we need norms (and sanctioning rules) to overspan multiple artifacts. This issue is left for future research.

4.3 Executing the Normative Dimension

In the previous section we have defined the syntax of the language by which the normative component of organizational artifacts can be specified and have shown an example of such an implementation. In this section we explain how the normative component of artifacts is executed by endowing the syntax with an operational semantics (Plotkin, 1981).

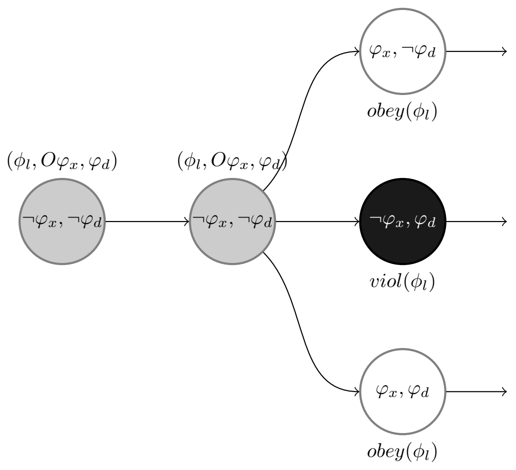
Intended Semantics of Obligations and Prohibitions

Before we proceed with defining the formal operational semantics of our norm enforcement mechanism, we first recapitulate and motivate the intended semantics of norm schemes and their instances, i.e. obligations and prohibitions. Once we have defined the formal operational semantics, we return to the intended semantics described in this section by proving that the norm schemes and their instances indeed behave as we claim. The semantics we attribute to norm instances is not completely designed by ourselves. In fact, the semantics of obligations is based on the one of Boella *et al.* (2008), in which the logic of conditional obligations with deadlines is presented. This choice is motivated by the fact that the way in which their semantics is presented is already close to an operational one. The choice for the semantics of prohibitions is mainly driven by some elementary properties from the field of deontic logic and pragmatical considerations.

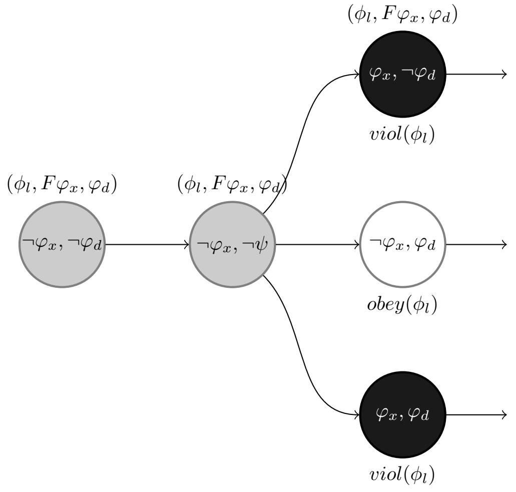
As discussed before, a norm scheme instantiates a norm instance (obligation or prohibition) when its condition is satisfied. We write a norm instance as a tuple $(\phi_l, \mathbb{M}\varphi_x, \varphi_d)$ with $\mathbb{M}$ either O to denote an obligation or F to denote a prohibition, ϕ_l the parameterized label identifying the norm scheme, φ_x the description of the state to be achieved or avoided before deadline φ_x. Once instantiated the behavior of an obligation and a prohibition differs as illustrated by figure 4.6.

Figure 4.6a shows the behavior of an instantiated obligation. As can be seen, the obligation stays into effect as long as it is not fulfilled or violated. An obligation is fulfilled when the state denoted by it can be entailed by the brute (and institutional) state before or when the deadline is reached. Fulfillment of an obligation labeled ϕ_l is marked by the assertion of a fact $obey(\phi_l)$ to the institutional state, whereas an institutional fact $viol(\phi_l)$ denotes a violation of ϕ_l. It is violated when φ_x has not been established from the state the obligation was instantiated up to the state in which the deadline can be entailed. Note that this means that when the brute state is changed such that deadline φ_d starts to hold and desired state φ_x starts to hold the obligation is considered to be fulfilled, instead of violated. From a conceptual point of view, one can argue that an action was undertaken to reach the desired state before the deadline held, and therefore the obligation should be considered achieved. The foremost argument, however, is of a technical nature. In this semantics we can express so called instantaneous obligations (see also (Broersen and van der Torre, 2007)). Instantaneous obligations are of the form $(\phi_l, O(\varphi_x), \top)$, i.e. φ_x should be established now. If we would have chosen a semantics in which an obligation is fulfilled if, and only if, it is reached before the deadline, an instantaneous obligation $(\phi_l, O(\varphi_x), \top)$ would be vacuously violated! Our semantics of an obligation resembles the one presented in (Boella et al., 2008).

The behavior of a prohibition is shown in figure 4.6b. Unlike an obligation, a prohibition stays into effect until its deadline φ_d is reached. That is, a prohibition is removed only when its deadline φ_d holds irrespective of whether it is violated.


$\varphi_x, \neg\varphi_d$
$obey(\phi_l)$
$(\phi_l, O\varphi_x, \varphi_d)$
$(\phi_l, O\varphi_x, \varphi_d)$
$\neg\varphi_x, \neg\varphi_d$
$\neg\varphi_x, \neg\varphi_d$
$\neg\varphi_x, \varphi_d$
$viol(\phi_l)$
φ_x, φ_d
$obey(\phi_l)$

(a) Obligation


$(\phi_l, F\varphi_x, \varphi_d)$
$\varphi_x, \neg\varphi_d$
$viol(\phi_l)$
$(\phi_l, F\varphi_x, \varphi_d)$
$(\phi_l, F\varphi_x, \varphi_d)$
$\neg\varphi_x, \neg\varphi_d$
$\neg\varphi_x, \neg\psi$
$\neg\varphi_x, \varphi_d$
$obey(\phi_l)$
φ_x, φ_d
$viol(\phi_l)$

(b) Prohibition

Figure 4.6: Some possible evolutions of an organizational artifact and the corresponding behavior of a norm instance $(\phi_l, \mathbb{M}(\varphi_x), \varphi_d)$ instantiated out of a norm scheme $\phi'_l : \langle \varphi'_c, \mathbb{M}(\varphi'_x), \varphi'_d \rangle$. Each circle denotes a possible state of the artifacts that evolves from one state into another by actions performed by agents. The relevant brute state is shown inside of the states, the effectuated norm instances above of them, and the institutional facts *viol* and *obey* below. States in which it is concluded that the norm instance is respected are colored white, whereas violation states are colored black. Gray states denote the situations where the norm instance is not yet violated or abided by.

Note that this implies that one and the same prohibition can be violated more than once. It is violated when the state described by φ_x can be entailed before or when the deadline starts to hold. It is abided by when φ_x cannot be entailed by the brute and institutional state during the period starting from when the prohibition was effectuated up to the moment the deadline holds for the first time. The fact that a prohibition ϕ_l has been respected is denoted by the assertion of a fact $obey(\phi_l)$, whereas a violation is denoted by an institutional fact $viol(\phi_l)$. Based on a similar reasoning as that for an obligation, we chose to consider an action that simultaneously establishes the prohibition's deadline φ_d as well as the undesired state φ_x as a violation. The action leading to φ_x was performed at a moment when establishing it was still forbidden. Moreover, we can now meaningfully express instantaneous prohibitions $(\phi_l, F(\varphi_x), \top)$ meaning that φ_x is forbidden only in the present state.

As observed before, a prohibition can in principle be violated more than once. Suppose, for example, that φ_x is established in multiple states until deadline φ_d. Then in each such state a violation as denoted by institutional fact $viol(\phi_l)$ can be inferred. To express a prohibition that disappears after it has been violated is to ensure that deadline $\varphi_d = \varphi_x$. In this case a violation will be detected because an action has been performed that led to a forbidden state, and because the deadline is now established, the prohibition will be revoked. Note, however, that this implies that the prohibition will be in effect as long as it is respected. Due to the lack of disjunctions in the deadline it is in general not possible to express a prohibition that stays into effect until its deadline $\varphi_d \neq \varphi_x$ is established or until it is violated.

Although a prohibition can be violated multiple times, it can only be fulfilled once. A prohibition is abided by when during the whole time the prohibition was in effect, the forbidden state φ_x is not established. To illustrate the intuition behind this consider, for example, we forbid someone to be in the park between sunset and sunrise. Suppose that someone walked through the park at night, but has left the park before sunrise. Would we now at sunrise – when the prohibition ceases to hold – conclude that this person has respected the prohibition? We would say no, the prohibition is only abided by if the person has not entered the park the whole night. In other words, once a prohibition is violated it cannot be abided by anymore. So, to conclude whether a prohibition is abided by or not, information about past violations becomes relevant.

Preliminaries

To understand the execution of the normative component, we extend the previously defined configuration of an organizational artifact to include norm schemes, norm instances and sanctioning rules also. However, before doing so, we provide some auxiliary functions regarding norm schemes and norm instances. They will facilitate defining the configuration of an organizational artifact and the transition rules explaining how an organizational artifact configuration may evolve.

We start with defining the notion of instantiating a norm scheme into a norm instance. As explained before, a norm instance is instantiated from its norm scheme when its condition is derivable from the brute and institutional state for some substitution of its formal parameters. Instantiating a norm scheme is then to apply this substitution on it resulting in a norm instance. Norm instances denoting a prohibition are annotated with extra information about past violations. This information is needed later on to enshrine obedience. For now, an annotation $\perp$ means that the prohibition has not been violated.

Definition 4.1 (Scheme Instantiation) *Let $ns = \phi_l(\overline{v_1}) : \langle \varphi_c(\overline{v_2}), \mathbb{M}(\varphi_x(\overline{v_3})), \varphi_d(\overline{v_4})\rangle$ be a norm scheme with $\mathbb{M}$ either an obligation O or prohibition F and $\overline{v_1}, \ldots, \overline{v_4}$ the sets of variables occurring in the formulae. Then the function $inst(ns, \theta)$ that instantiates a norm instance from norm scheme ns given a formal substitution θ for the variables is defined as:*

$$inst(ns, \theta) = \begin{cases} (\phi_l(\overline{v_1})\theta, O(\varphi_x(\overline{v_3})\theta), \varphi_d(\overline{v_4})\theta) & \text{if } \mathbb{M} = O \\ (\phi_l(\overline{v_1})\theta, F(\varphi_x(\overline{v_3})\theta), \varphi_d(\overline{v_4})\theta) \circ \perp & \text{if } \mathbb{M} = F \end{cases}$$

The substitution that is applied in instantiating a norm scheme is obtained via its condition. In what follows we demand norm instances to be ground, i.e. no variables to occur in them. To ensure a norm instance to be ground, we demand each variable that occurs in the norm scheme also to occur in the condition. Having unground norm instances raises the question whether the variables occurring in them are existentially or universally quantified. This question and a possible extension of the language to include quantifiers is left for future research. To ensure norm instances to be ground, we demand each variable that occurs in the norm scheme also to occur in its condition. This is formally defined by the next definition introducing well-formedness of norm schemes.

Definition 4.2 (Well-formedness of Norm Schemes) *Given a norm scheme ns of the form $\phi_l(\overline{v_1}) : \langle \varphi_c(\overline{v_2}), \mathbb{M}(\varphi_x(\overline{v_3})), \varphi_d(\overline{v_4})\rangle$ such that $\mathbb{M}$ either an obligation O or prohibition F and $\overline{v_1}, \ldots, \overline{v_4}$ the sets of variables occurring in the formulae. We say that ns is well-formed if and only if it holds that $\overline{v_1} \cup \overline{v_3} \cup \overline{v_4} \subseteq \overline{v_2}$.*

Having defined some auxiliary functions and knowing how (well-formed) norm schemes are instantiated we are now in a position to define the configuration of an organizational artifact with a normative component. This definition extends the previous definition of an organizational artifact as defined in chapter 3. For the sake of readability we repeat all the previously defined components here.

Definition 4.3 (Artifact Configuration) *An organizational artifact, typically denoted by O, is defined as a tuple $\langle id, \sigma_b, \sigma_i, \mathrm{E}, \Delta, \delta, \Sigma, \epsilon_{in}, \epsilon_{out}\rangle$, in which:*

- *id is a name uniquely identifying the artifact;*

- σ_b *is a set of ground, first-order atoms describing the brute state of the artifact;*
- σ_i *is a set of ground, first-order atoms describing the artifact's institutional state;*
- E *is a set of effect specifications;*
- Δ *a set of well-formed norm schemes;*
- $\delta \subseteq \{inst(ns, \theta) \mid ns \in \Delta \text{ and } \theta$ *a ground substitution of the variables*$\}$*, i.e. a set of ground norm instances defining the active obligations and prohibitions;*
- Σ *the set of sanctioning rules by which the punishments and rewards associated to norm compliance and violation are defined;*
- ϵ_{in} *is a list of ground first-order atoms, the events perceived by the artifact;*
- ϵ_{out} *is a list of ground first-order atoms either with predicate name Send or Do, the communication messages to be sent and external actions to be performed.*

Just like in previous chapter we define the notion of an initial artifact configuration. This configuration redefines the one previously defined in chapter 3. The initial organizational artifact is the one that is determined by the program code. The artifact's program defines the artifact's name, its initial brute state, the effect specifications, its norm schemes and their associated sanctions. Initially, no events are received, no events need to be sent, no obligations and prohibitions have been instantiated, and no norms have been complied with or violated and no roles have been enacted. This is defined by the following definition.

Definition 4.4 (Initial Artifact Configuration) *An initial artifact configuration is an artifact* $\langle id, \sigma_b, \emptyset, \mathrm{E}, \Delta, \emptyset, \Sigma, \emptyset, \emptyset \rangle$ *specified by a program such that id is specified by the program's name component,* σ_b *is characterized by the program's facts component,* E *is defined by the program's effect component,* Δ *is determined by the program's norm component and* Σ *is defined by the program's sanctions.*

Execution of Norm Schemes and Their Instances

Having defined the organization configuration we can now define the transition rules that specify how an organizational artifact may evolve in the light of norm schemes that trigger obligations and prohibitions, and monitoring of the compliance of these instantiated norm schemes. In what follows we assume an artifact configuration $\langle id, \sigma_b, \sigma_i, \mathrm{E}, \Delta, \delta, \Sigma, \epsilon_{in}, \epsilon_{out} \rangle$. The components that will not change during computation will be omitted, viz. the artifact's *id*, its effects specification E, its norm schemes Δ and its sanctioning rules Σ.

We start with the transition rule explaining how norm schemes are triggered. Call to mind that a norm scheme is triggered when its condition can be entailed by the artifact's brute and institutional state. Triggering a norm instance is to create a norm instance (as defined by the instantiation function) which is added to the set of norm instances. Triggering is defined by the next transition rule.

Transition Rule *Given ground substitution θ, the rule for triggering of norm schemes is defined as:*

$$\frac{ns = (\phi_l : \langle \varphi_c, \mathbb{M}(\varphi_x), \varphi_d \rangle) \quad ns \in \Delta \quad \sigma_b \cup \sigma_i \models \varphi_c\theta \quad ni = inst(ns, \theta)}{\langle \sigma_b, \sigma_i, \delta, \epsilon_{in}, \epsilon_{out} \rangle \longrightarrow_{org} \langle \sigma_b, \sigma_i, \delta \cup \{ni\}, \epsilon_{in}, \epsilon_{out} \rangle} \quad (trig)$$

Next, we define the rules specifying the (non)compliance of an obligation. When there is an obligation $(\phi_l, \varphi_c, O(\varphi_x), \varphi_d)$ in a certain configuration and the deadline φ_d has passed, but φ_x has not been achieved yet then this obligation is violated. Remember that violated obligations are removed from the set of norm instances. An obligation is achieved in this configuration when the state denoted by φ_x can be entailed by the brute and institutional state. Also in this case the obligation is removed from the set of norm instances. This is defined by the following two transition rules. The first pertains to violation, the second to fulfillment of an obligation.

Transition Rule *The rule for violation of an obligation is defined as follows:*

$$\frac{ni = (\phi_l, O(\varphi_x), \varphi_d) \quad ni \in \delta \quad \sigma_b \cup \sigma_i \not\models \varphi_x \quad \sigma_b \models \varphi_d}{\langle \sigma_b, \sigma_i, \delta, \epsilon_{in}, \epsilon_{out} \rangle \longrightarrow_{org} \langle \sigma_b, \sigma_i \cup \{viol(\phi_l)\}, \delta \setminus \{ni\}, \epsilon_{in}, \epsilon_{out} \rangle} \quad (viol_o)$$

Transition Rule *The rule for fulfillment of an obligation is defined as follows:*

$$\frac{ni = (\phi_l, O(\varphi_x), \varphi_d) \quad ni \in \delta \quad \sigma_b \cup \sigma_i \models \varphi_x}{\langle \sigma_b, \sigma_i, \delta, \epsilon_{in}, \epsilon_{out} \rangle \longrightarrow_{org} \langle \sigma_b, \sigma_i \cup \{obey(\phi_l)\}, \delta \setminus \{ni\}, \epsilon_{in}, \epsilon_{out} \rangle} \quad (obey_o)$$

Just like obligations we define the transition rules that specify when a prohibition is (not) abided by. When there is a prohibition $(\phi_l, \varphi_c, F(\varphi_x), \varphi_d)$ in a certain configuration and the deadline φ_d has not passed, but φ_x has been achieved, then this prohibition is violated. As opposed to an obligation, a violated prohibition remains in effect. That is, it is not removed from the set of norm instances. A prohibition is abided by when deadline φ_d holds and φ_x has not been established during the whole period for which the prohibition was in effect. In defining the transition rules by which we can derive a transition $\gamma_k \to \gamma_{k+1}$ we cannot refer to the trace $\gamma_0 \to^* \gamma_{k-1}$ that was used to reach γ_k. But in defining the transition rule for obedience of a prohibition we still need to know if the prohibition was violated in the past. Recall that, to have this information at our disposal in defining the transition for obedience, we annotate a prohibition ni with information about past violations by adding flags $\perp$ (not violated) and $\top$ (violated). In concrete, we write $ni \circ \top$ to denote that prohibition ni has been violated, whereas annotation $ni \circ \perp$ means that ni has been abided by up to now. Note that obligations do not need this annotation, because for detecting the fulfillment of an obligation we only need the present state (cf. transition rule $obey_o$).

The following two transition rules define when a prohibition is not complied with. The first rules amounts to the case in which a prohibition is violated before its deadline holds. In this case the prohibition will stay in effect as its deadline is not yet reached, i.e. it is not removed from the set of norm instances. Besides asserting a violation fact to denote its violation, also its flag is set to $\top$ to record that it has been violated (and remember this during the rest of its lifetime). The second rule pertains to the case in which a prohibition is violated just when the deadline holds. In this case it is removed and a violation fact is added to the institutional state.

Transition Rules *The rules for violation of a prohibition are defined as follows:*

$$\frac{ni = (\phi_l, \varphi_c, F(\varphi_x), \varphi_d) \quad (ni \circ s) \in \delta \quad \sigma_b \cup \sigma_i \models \varphi_x \quad \sigma_b \not\models \varphi_d}{\langle \sigma_b, \sigma_i, \delta, \epsilon_{in}, \epsilon_{out} \rangle \longrightarrow_{org} \langle \sigma_b, \sigma_i \cup \{viol(\phi_l)\}, (\delta \setminus \{ni \circ s\}) \cup \{ni \circ \top\}, \epsilon_{in}, \epsilon_{out} \rangle} \quad (viol1_p)$$

$$\frac{ni = (\phi_l, \varphi_c, F(\varphi_x), \varphi_d) \quad (ni \circ s) \in \delta \quad \sigma_b \cup \sigma_i \models \varphi_x \quad \sigma_b \models \varphi_d}{\langle \sigma_b, \sigma_i, \delta, \epsilon_{in}, \epsilon_{out} \rangle \longrightarrow_{org} \langle \sigma_b, \sigma_i \cup \{viol(\phi_l)\}, \delta \setminus \{ni \circ s\}, \epsilon_{in}, \epsilon_{out} \rangle} \quad (viol2_p)$$

The following two transition rules handle the case in which the deadline holds, but the prohibition is abided by, at least at that moment. More specifically, the first transition rule pertains to the case in which the deadline holds of a prohibition that is momentarily not violated and has not been violated in the past (as denoted by its annotation $\perp$.) In this case the prohibition is removed from the set of norm instances and an institutional fact recording its obedience is asserted to the institutional facts. When this prohibition was not respected in the past, i.e. it has been violated before (as indicated by its annotation $\top$), such an obedience fact is not added to the institutional state. This is expressed by the second transition rule.

Transition Rules *The rules for detecting compliance to a prohibition at the deadline are defined as follows:*

$$\frac{ni = (\phi_l, \varphi_c, F(\varphi_x), \varphi_d) \quad ni \circ \perp \in \delta \quad \sigma_b \cup \sigma_i \not\models \varphi_x \quad \sigma_b \models \varphi_d}{\langle \sigma_b, \sigma_i, \delta, \epsilon_{in}, \epsilon_{out} \rangle \longrightarrow_{org} \langle \sigma_b, \sigma_i \cup \{obey(\phi_l)\}, \delta \setminus \{ni \circ \perp\}, \epsilon_{in}, \epsilon_{out} \rangle} \quad (obey1_p)$$

$$\frac{ni = (\phi_l, \varphi_c, F(\varphi_x), \varphi_d) \quad ni \circ \top \in \delta \quad \sigma_b \cup \sigma_i \not\models \varphi_x \quad \sigma_b \models \varphi_d}{\langle \sigma_b, \sigma_i, \delta, \epsilon_{in}, \epsilon_{out} \rangle \longrightarrow_{org} \langle \sigma_b, \sigma_i, \delta \setminus \{ni \circ \top\}, \epsilon_{in}, \epsilon_{out} \rangle} \quad (obey2_p)$$

As can be seen from the definition of above transition rules, the normative judgment about the organizational artifact, i.e. information about (non)conformity to the norms, is stored by the institutional facts. As we shall see in a moment, sanctioning rules are applied based on these institutional facts to punish infringements and reward obedience. The institutional facts are designed such that, by default, they do not contain any information about when a norm has been violated or abided by. The intended meaning of violation (and obedience) facts is that a norm is violated (or abided) by *now*. That is to say, no information is stored in the institutional facts about past (non)conformance. As we will show later on, the organizational coordination cycle is designed such that, before detecting new infringements and obedience, all institutional facts containing information about (non)conformance are removed first. For this purpose, we introduce a transition rule that takes care of cleaning up the institutional facts by removing all information concerning the normative judgment. Note that the *rea* facts keeping information about role enactments should not be removed.

Transition Rule *Let pred(ϕ) be a function evaluating to the predicate name of a first-order atom ϕ. Then the rule for cleaning up institutional facts is defined as follows:*

$$\frac{}{\langle \sigma_b, \sigma_i, \delta, \epsilon_{in}, \epsilon_{out} \rangle \longrightarrow_{org} \langle \sigma_b, \sigma_i', \delta, \epsilon_{in}, \epsilon_{out} \rangle} \qquad (clean)$$

where $\sigma_i' = \sigma_i \setminus \{\phi \mid \phi \in \sigma_i \text{ and } pred(\phi) \in \{viol, obey\}\}$

When, for some reason, past occurrences of non(conformity) should be recorded, for example, to apply sanctions based on past and current violations, this can be achieved by designing the norm schemes such that information about when the norm instance was in effect is stored by the parameters of the norm scheme's label. Of course, this implies leaving out the previous rule in the organizational coordination cycle. Alternatively, information about past (non)conformities could be stored by the brute facts that are established by the sanctioning rules.

Executing Sanctioning Rules

Now we know when a norm has been complied with or has been violated, we define the transition rule that explains how sanctions are applied by executing the sanctioning rules. Recall that a sanctioning rule is applicable when its antecedent is satisfied by the brute and institutional state for some substitution θ. To apply a sanctioning rule is to assert the facts of its consequent to the brute state of the organizational artifact. That is, if the special atom `false` is not part of the sanctioning rule's consequent. Remember that similar to effect specifications, the consequent of a sanctioning rule may contain special atoms *Send* and *Do* for sending messages and performing actions, both of which are not asserted to the brute state, but added to the list of out events instead. Recall that we use a colon ':' to denote the operation of appending two lists.

Transition Rule *The rule for applying an enforced sanctioning rule is defined as follows:*

$$\frac{(\varphi \texttt{=>} \Psi) \in \Sigma \quad \sigma_b \cup \sigma_i \models \varphi\theta \quad \texttt{false} \notin \Psi}{\langle \sigma_b, \sigma_i, \delta, \epsilon_{in}, \epsilon_{out} \rangle \longrightarrow_{org} \langle \sigma_b', \sigma_i, \delta, \epsilon_{in}, \epsilon_{out}' \rangle} \qquad (sanc)$$

$$\begin{aligned} \text{where } \sigma_b' &= \sigma_b \uplus \{\phi \mid \phi \in \Psi\theta \text{ and } pred(\phi) \notin \{Send, Do\}\} \\ \epsilon_{out}' &= \epsilon_{out} : [\phi \mid \phi \in \Psi\theta \text{ and } pred(\phi) \in \{Send, Do\}] \end{aligned}$$

The above transition rule denotes the application of a sanctioning rule pertaining to an enforcement strategy of the norms. In other words, the violation of some norm(s) is tolerated, but possibly punished. In case a sanctioning rule is applicable that pertains to a regimentation strategy of some norm(s), i.e. has special atom `false` in its consequent, the sanction is not materialized. In fact, not only the sanctioning rule is not effectuated, the whole configuration that causes this intolerable situation is dismissed. Note that this is not expressible by one single transition rule, because we cannot refer to the artifact configuration(s) preceding the undesirable configuration that has just been reached by the application of some transition rule(s). We can, however, do exactly that in our coordination strategy language in which we are able to talk about the outcome of a sequence of transitions. To enable us to refer to intolerable configurations reached by the application of some "regimented sanctioning rule" in defining the organizational coordination strategy, we define the following transition rule. A configuration is intolerable if some sanctioning rule with special atom `false` in its consequent is applicable. To indicate the impossibility of actually applying a sanctioning rule with `false` in its consequent, we write $\perp$ to denote a "deadlock" configuration. As we shall see later on, the coordination strategy ensures that such a state is never reached.

Transition Rule *The rule for applying a regimented sanctioning rule is defined as follows:*

$$\frac{(\varphi \texttt{=>} \Psi) \in \Sigma \quad \sigma_b \cup \sigma_i \models \varphi\theta \quad \texttt{false} \in \Psi\theta}{\langle \sigma_b, \sigma_i, \delta, \epsilon_{in}, \epsilon_{out} \rangle \longrightarrow_{org} \perp} \qquad (reg)$$

When at some point a regimented sanctioning rule is applicable, we want to roll back to the desirable configuration preceding the intolerable one we just reached. Typically, the effect of some action has been determined (followed by triggering, monitoring and sanctioning), causing the intolerable configuration. We thus want to return to the configuration before the effect of the action was effectuated. However, because at this point we already know that handling the received action would lead to a regimented action, there is no sense in leaving this action on top of the list of received actions when rolling back. Therefore, we introduce the following, auxiliary transition rule that merely removes the head of the list of received actions. We use this transition rule in defining our coordination strategy that accounts for regimentation.

Transition Rule *Let α be an action (including enact and deact) performed by an agent. Then the rule for ignoring this action is defined as follows:*

$$\frac{}{\langle \sigma_b, \sigma_i, \delta, [\alpha] : \epsilon_{in}, \epsilon_{out} \rangle \longrightarrow_{org} \langle \sigma_b, \sigma_i, \epsilon_{in}, \epsilon_{out} \rangle} \qquad (ignact)$$

4.4 Organizational Coordination Strategies

Just like previous chapter we only elaborated on the meaning of the programming constructs, without committing to a particular sequence in which they are executed. Different sensible strategies exist, each possibly yielding different outcomes. In this section we discuss some possible organizational coordination strategies by means of the coordination language introduced in previous chapter. The next section, that formally studies some of the elementary properties of the normative component, further elaborates upon the implications different strategies may have.

Call to mind that the strategy language we borrowed from Astefanoaei *et al.* and Eker *et al.* (2009a; 2007) includes: construct $l@\gamma$ for applying a transition rule labeled l to a configuration γ, evaluating to the set of all possible configurations of applying this transition rule; $idle@\gamma$ evaluating to $\{\gamma\}$ and $fail@\gamma$ evaluating to the empty set; sequence operator ';' for sequencing expressions; choice operator | for non-deterministically choosing among two expressions; operator ! for repeating an expression until no longer applicable; operator + to repeat an expression one or more times; operator $*$ to repeat an expression zero or more times; and choice operator of the form $E\ ?\ E'\ :\ E''$ to mean that if expression E does not result in the empty set we apply expression E' on its outcome, otherwise E'' on the initial configuration. In what follows we apply some of the strategies that were also discussed by Astefanoaei *et al.* (2009a) for the counts-as based language on the normative language presented here.

4.4.1 Monitoring and Triggering Strategies

We start with some strategy that focus on the triggering and monitoring process, and do not include sanctioning. We do so, because sanctions complicate the proofs of the propositions we posit next (see also (Tinnemeier et al., 2009c)). As a first strategy consider the following organizational coordination step (that presumably comes after having determined the effect of an action):

$$clean; (trig!; (viol_o|viol1_p|viol2_p)!; (obey_o|obey1_p|obey2_p)!)! \qquad \text{(totalitarian)}$$

saying that we first clean information about past violations and obedience, after which we trigger *all* applicable norm schemes, detect *all* violations of obligations and prohibitions, and finally detect *all* obedience of obligations and prohibitions.

The process of triggering and checking for compliance and infringement is repeated, because the violation of or compliance to some norm instance may trigger a secondary norm scheme. This coordination strategy is called totalitarian, because it amounts to a strategy where *all* norms are enforced.

Remark that $trig!; (viol_o|viol1_p|viol2_p)!; (obey_o!|obey1_p|obey2_p)!$ always terminates, because the sets of brute and institutional facts and the set of norm schemes are all finite and, consequently, will create a finite set of norm instances. However, it should also be remarked that the repetition of the triggering and monitoring sequence might not terminate at all. Suppose, for example, we have a norm scheme that instantiated an obligation that is instantaneously fulfilled (say), and thus removed. Because the same norm scheme then might again instantiate the same obligation this process would be repeated ad infinitum. Instead of accounting for such situations in the semantics and strategies, it is the responsibility of the programmer to define the schemes' conditions such that this situation never happens. In (Tinnemeier et al., 2009c) we stratified the norms in different levels such that at level 0 we find norms of which the condition only refers to brute facts, and at each level $k > 0$ we find norms that only trigger because of some violation or fulfillment of a level $k - 1$ norm. Determining (non)conformity and triggering is then done per norm level such that it will always terminate.

In the totalitarian strategy every norm is enforced. That is to say, if a norm scheme is applicable, it will be applied and, consequently, all instances it may create will be instantiated. Moreover, each norm instance is monitored for violation and fulfillment. When there is a large amount of norm schemes creating many norm instances, a totalitarian strategy might not be feasible. As an alternative, a strategy might be deployed in which not all instantiable norm instances are actually created, but only some of them. An example of such a strategy is the following:

$$clean; (trig^*; (viol_o|viol1_p|viol2_p)!; (obey_o|obey1_p|obey2_p)!)^+ \qquad \text{(liberal)}$$

that imposes no restrictions on if and how many norm schemes are triggered, i.e. not all norms are enforced. The process of triggering and checking for compliance and infringement is possibly repeated (due to the repetition operator $+$), because the violation of or compliance to some norm instance may trigger a secondary norm scheme. It should be emphasized that also in this strategy *all* active obligations and prohibitions are monitored for (non)conformance. This is an important feature in proving the propositions that follow.

Of course, many other sensible coordination strategies are possible, but a thorough investigation of all these strategies is not our aim. By showing these we merely want to point out that different strategies do exist. Our foremost aim is to show that in the coordination strategies *totalitarian* and *liberal*, the interpretation of obligations and prohibitions is indeed the one as explained in 4.6. The crux of most of the proofs that will follow is that in both strategies *all* norm instances are monitored for (non)compliance and that this is always done *after* triggering norm schemes.

Remember the coordination step 'action' we defined in previous chapter for processing the effect of an action and the coordination step 'out' for processing all the events (messages and actions) that are stored in the list of out events. In what follows we assume an organizational coordination strategy E that is either of the form action; out; totalitarian or action; out; liberal. Recall the function $C^n_E(\mathrm{O})$ (defined in chapter 3) that evaluates to the execution trace that results after iteratively applying coordination strategy E n times on organizational artifact configuration O.

What is more, we write $\sigma_b[j]$ to refer to the brute state of the jth organizational configuration, i.e. O_j, as part of a coordinated trace $\mathrm{O}_0 \cdots \mathrm{O}_n$. We use a similar notation to refer to the other elements of O_j. Although it should be clear from the context, the notation we use for organizational artifacts, viz. O, should not be confused with the modality for obligations, namely an italicized O.

The first three propositions are about the behavior of an obligation. The first proposition shows that an obligation persists as long as it is not achieved and it is still before the deadline. More precisely, it will be in effect if and only if it is not violated or achieved.

Proposition 4.5 *Let O_0 be an artifact configuration s.t. $ni = (\phi_l, O(\varphi_x), \varphi_d) \in \delta[0]$ and $C^n_E(\mathrm{O}_0) = \mathrm{O}_0 \cdots \mathrm{O}_n$ with $n > 0$ be a coordinated trace. Then it holds that $ni \in \delta[n]$ if and only if $\sigma_b[j] \cup \sigma_i[j] \not\models \varphi_x$ and $\sigma_b[j] \not\models \varphi_d$ for $0 < j \leq n$.*

Proof
($\leftarrow$) Given a derivation of length 1 we have to prove that $ni \in \delta[1]$. The totalitarian strategy ensures that $ni \in \delta[1]$ because: 1) $ni \in \delta[0]$ (by assumption); 2) $viol_o$ and $obey_o$ are the only rules that remove obligations; 3) both rules are not applicable for $(\phi_l, O(\varphi_x), \varphi_d)$ because $\sigma_b[1] \cup \sigma_i[1] \not\models \varphi_x$ and $\sigma_b[1] \not\models \varphi_d$ (by assumption). The proof for the inductive case proceeds by a similar reasoning.
($\rightarrow$) Given a derivation of length 1 we have to prove that $\sigma_b[1] \cup \sigma_i[1] \not\models \varphi_x$ and $\sigma_b[1] \not\models \varphi_d$. In the strategy E this is indeed the case, because: 1) $ni \in \delta[1]$ (by assumption); 2) transition rule trig is applied before $viol_o$ and $obey_o$ are applied until no longer applicable; and 3) ni would have been removed by $viol_o$ or $obey_o$ if either $\sigma_b[1] \cup \sigma_i[1] \models \varphi_x$ or $\sigma_b[1] \models \varphi_d$. The proof for the inductive case proceeds by a similar reasoning. ∎

An obligation of which the state denoted by it has not been fulfilled from the moment it was created up to the deadline is marked violated. It is marked fulfilled when the state denoted by it is achieved before or at the state at which the deadline first holds. This is demonstrated by the proposition below. Note that from previous proposition it follows that an obligation is removed upon a violation or fulfillment.

Proposition 4.6 *Let O_0 be an artifact configuration s.t. $ni = (\phi_l, O(\varphi_x), \varphi_d) \in \delta[0]$. Then for every coordinated trace $C^n_E(\mathrm{O}_0) = \mathrm{O}_0 \cdots \mathrm{O}_n$ for $n > 0$ s.t. $\sigma_b[j] \cup \sigma_i[j] \not\models \varphi_x$ and $\sigma_b[j] \not\models \varphi_d$ for $0 < j < n$ it holds that:*

(a) if $\sigma_b[n] \cup \sigma_i[n] \not\models \varphi_x$ and $\sigma_b[n] \models \varphi_d$ then $\sigma_i[n] \models viol(\phi_l)$

(b) if $\sigma_b[n] \cup \sigma_i[n] \models \varphi_x$ then $\sigma_i[n] \models obey(\phi_l)$

Proof
(a) Given a derivation of length 1 we have to prove that $\sigma_i[1] \models viol(\phi_l)$. The coordination strategy ensures that $\sigma_i[1] \models viol(\phi_l)$, because: 1) $ni \in \delta[0]$ (by assumption); 2) $\sigma_b[1] \cup \sigma_i[1] \not\models \varphi_x$ and $\sigma_b[1] \models \varphi_d$ (by assumption); 3) transition rule $viol_o$ will be applied (because of strategy E and 1 and 2); transition rule $viol_o$ ensures that $\sigma_i[n] \models viol(\phi_l)$. Assume this proposition holds for a derivation of length k. To prove that it also holds for a derivation of length $k+1$. That $ni \in \delta[k]$ follows from proposition 4.5. The rest of the proof is based on a similar reasoning as the base case.
(b) Given a coordinated trace of length 1 we have to prove that $\sigma_i[1] \models viol(\phi_l)$. The strategy E ensures that $\sigma_i[1] \models obey(\phi_l)$, because: 1) $ni \in \delta[0]$ (by assumption); 2) $\sigma_b[1] \cup \sigma_i[1] \models \varphi_x$ (by assumption); 3) transition rule $obey_o$ will be applied (because of strategy E and 1 and 2); transition rule $obey_o$ ensures that $\sigma_i[n] \models obey(\phi_l)$. Assume this proposition holds for a derivation of length k. To prove that it also holds for a trace of length $k+1$. That $ni \in \delta[k]$ follows from proposition 4.5. The rest of the proof is based on a similar reasoning as the base case. ■

Unlike an obligation, a prohibition is removed if and only if the deadline is reached, irrespective of whether the prohibition is violated or not. This is demonstrated by the proposition that follows next.

Proposition 4.7 *Let O_0 be an artifact configuration s.t. $ni = (\phi_l, F(\varphi_x), \varphi_d)$ such that $ni \circ s \in \delta[0]$. Then for every coordinated trace $C_E^n(O_0) = O_0 \cdots O_n$ for $n > 0$ it holds that there exists an s' that is either $\top$ or $\bot$ such that $ni \circ s' \in \delta[n]$ if and only if $\sigma_b[j] \not\models \varphi_d$ for $0 \leq j \leq n$.*

Proof
($\leftarrow$) Given a derivation of length 1 we have to prove that $ni \circ s' \in \delta[1]$. The coordination strategy ensures that $ni \circ s' \in \delta[1]$ because: 1) $ni \in \delta[0]$ (by assumption); 2) $viol2_p$, $obey1_p$ and $obey2_p$ are the only rules that remove prohibitions; 3) neither of these rules is applicable for ni because $\sigma_b[1] \not\models \varphi_d$ (by assumption). The proof for the inductive case proceeds by a similar reasoning.
($\rightarrow$) Given a derivation of length 1 we have to prove that $\sigma_b[1] \not\models \varphi_d$. In the strategy E this is indeed the case, because: 1) $ni \circ s' \in \delta[1]$ (by assumption); 2) transition rule trig is applied before $viol2_p$, $obey1_p$ and $obey2_p$ are applied until no longer applicable; and 3) $ni \circ s'$ would have been removed by either $viol2_p$, $obey1_p$ or $obey2_p$ if $\sigma_b[1] \models \varphi_d$. The proof for the inductive case proceeds by a similar reasoning. ■

A prohibition is violated whenever the forbidden state denoted by it is established before or at the moment at which the deadline holds. It is abided by if its forbidden state has not been established during the whole period the prohibition was in effect. These two properties are shown by the next proposition.

Proposition 4.8 *Let O_0 be an artifact configuration s.t. $ni = (\phi_l, F(\varphi_x), \varphi_d)$ and $ni \circ \bot \in \delta[0]$. Then for every coordinated trace $C_E^n(\mathrm{O}_0) = \mathrm{O}_0 \cdots \mathrm{O}_n$ for $n > 0$ s.t. $\sigma_b[j] \not\models \varphi_d$ for $0 \leq j < n$ it holds that:*

(a) if $\sigma_b[n] \cup \sigma_i[n] \models \varphi_x$ then $\sigma_i[n] \models viol(\phi_l)$.

(b) if $\sigma_b[j] \cup \sigma_i[j] \not\models \varphi_x$ for $0 \leq j \leq n$ and $\sigma_b[n] \models \varphi_d$ then $\sigma_i[n] \models obey(\phi_l)$

Proof
(a) Given a trace of length 1 we have to prove that $\sigma_i[1] \models viol(\phi_l)$. The strategy E ensures that $\sigma_i[1] \models viol(\phi_l)$, because: 1) $ni \circ \bot \in \delta[0]$ (by assumption); 2) $\sigma_b[1] \cup \sigma_i[1] \models \varphi_x$ (by assumption); 3) either transition rule $viol1_p$ or $viol2_p$ will be applied (because of strategy and 1 and 2); both transition rules ensure that $\sigma_i[n] \models viol(\phi_l)$. Assume this proposition holds for a trace of length k. To prove that it also holds for a derivation of length $k + 1$. That $ni \circ s \in \delta[k]$ with s either $\top$ or $\bot$ follows from proposition 4.7. The rest of the proof is based on a similar reasoning as the base case.
(b) Given a trace of length 1 we have to prove that $\sigma_i[1] \models obey(\phi_l)$. The strategy E ensures that $\sigma_i[1] \models obey(\phi_l)$, because: 1) $ni \circ \bot \in \delta[0]$ (by assumption); 2) $\sigma_b[1] \models \varphi_d$ and $\sigma_b[1] \cup \sigma_i[1] \not\models \varphi_x$ (by assumption); transition rule $obey1_p$ will be applied (because of strategy and 1 and 2); transition rule $obey1_p$ ensures that $\sigma_i[1] \models obey(\phi_l)$. Assume this proposition holds for a derivation of length k. To prove that it also holds for a trace of length $k + 1$. Observe that $ni \circ \bot \in \delta[k]$ because of: 1) proposition 4.7 and the fact that rule $viol1_p$ will never be applied because of the assumption $\sigma_b[j] \cup \sigma_i[j] \not\models \varphi_x$ for $0 \leq j \leq n$. From here on, the rest of the proof is similar as that of the base case. ∎

4.4.2 Sanctioning Strategies

Up till now we did not mention sanctions that are applied upon a violation or conformance to a norm. In our framework there are basically two ways to respond to a violation of some norm. The first is to impose punishments on the brute state, whereas the second is to rollback to the situation the artifact was in before the effect of the action causing the violation has been applied. The first strategy pertains to an enforcement of the norms, whereas the latter pertains to the regimentation of the norms (see also chapter 2).

Choosing an enforcement strategy does not exclude the possibility to use regimentation also. Typically, only some norms will be regimented, whereas most norms will be enforced. Whichever strategy is chosen, still many choices remain.

For example, whether to apply every applicable sanctioning rule or to apply only some of them. What is more, different sanctioning strategies can be applied in combination with different monitoring and triggering strategies. Consider, for example, the next coordination strategy:

$$\mathit{test}(\text{action}; \text{totalitarian}; \mathit{reg}) \ ? \ \mathit{ignact} \ : \ \text{totalitarian}; \mathit{sanc}!; \mathit{out}! \tag{totalitarian-reg}$$

saying that a totalitarian strategy will be applied after which all applicable sanctioning rules will be effectuated only if applying the totalitarian strategy does not enable the application of regimentation rules. If a regimented sanctioning rule is indeed applicable the result of this strategy is as if the action would not have been received at all. Observe that this implies that no punishments are carried out. Also note that in case an organizational artifact would not have any regimented sanctioning rules, this strategy would produce the same outcome as the strategy:

$$\text{action}; \text{totalitarian}; \mathit{sanc}!; \mathit{out}! \tag{totalitarian-enf}$$

In both of these strategies the totalitarian strategy could be easily replaced by a liberal strategy. Moreover, instead of applying all applicable sanctioning rules we could have chosen to apply only some. Again, the investigation of these and more alternative strategies is beyond the scope of present work. Though, it is interesting to note that in these strategies the propositions 4.5–4.8 do not hold. This is due to the fact that the application of a sanction might establish or undo the φ_d and φ_x components of a norm instance $(\phi_l, \mathbb{M}(\varphi_x), \varphi_d)$. Suppose, for example, we have a norm instance $ni = (\phi_l, O(\varphi_x), \varphi_d)$ for some artifact configuration, and that φ_x can be entailed by the brute state. Applying rule obey_o will then remove this norm instance and mark it as achieved. Now, suppose that some sanctioning rule is applied such that φ_x no longer holds, but ni is still removed and considered achieved. For those who think that such interference should be prevented consider the following example. In the conference management system example we regimented the act of uploading a paper exceeding the page limit. Now suppose that we still want to prevent papers of more than fifteen pages from being uploaded, but we also want to punish one for trying (regimentation is thus not an option). Then we could model this by replacing the original regimented sanctioning rule by a rule (with the same antecedent) that removes the paper from the system and at the same time imposes a sanction. The facts that are imposed as sanction thus interfere with the state denoted by the prohibition.

4.5 On the Notion of Normative Conflict and Coherency

Due to the many different representations and meanings of normative concepts that have been proposed in literature, there are also many different conceptions of when a set of norms is said to be conflicting. An overview of some different notions of normative conflict can be found in the article of Elhag *et al.* (2000). Standard deontic logic, for example, comprises $\neg(Op \wedge O\neg p)$ and $\neg O(p \wedge \neg p)$ as theorems. The first stating that one cannot be obliged to do conflicting action, and the second stating that there cannot be contradictory obligations. In standard deontic logic, normative conflicts thus do not exist. However, deriving (by law) that someone is obliged to pay his taxes and is not obliged to pay his taxes seems strange, but might happen in practice (and is why we have judges to interpret the law.) Based on the observation that such situations seem to occur in practice, some have argued that normative conflicts do exist (cf. (Hansen et al., 2007) for a discussion). That normative conflicts may occur, does not imply, however, we should do nothing to prevent them. Indeed, in (Vasconcelos et al., 2007) Vasconcelos *et al.* propose a mechanism for avoiding normative conflicts. They adopt a representation of norms expressing obligations, prohibitions and permission relating to a single action. They mark a situation in which an agent is simultaneously obliged and forbidden to perform an action α as a normative conflict.

Suppose we adopt the same definition as that of (Vasconcelos et al., 2007), meaning that we say that two norms are in conflict when whatever action is undertaken, a violation is inevitable. Because the situations that should be achieved or avoided and the deadlines of our norms range over states that can be achieved, defining when a set of norm instances is conflicting is suddenly not trivial anymore. It seems, for example, that we should mark two norm instances $(l_1, F(p), q)$ and $(l_2, O(p), q)$ as conflicting, because a violation is inevitable. However, here a violation is only inevitable in case the deadline q will eventually hold. A more fundamental difficulty in defining a notion of normative conflict is caused by the fact that both deadline and the state denoted by the norm instances may be related. Consider, for example, three norm instances $(l_1, O(p), q)$, $(l_2, O(r), p)$ and $(l_3, F(r), q)$. To abide by the first obligation we should establish p before q, but to also abide by the second obligation we should have first achieved r. This implies that we achieve r before q, but this would violate the prohibition! It somehow seems that from the norm instances $(l_1, O(p), q)$ and $(l_2, O(r), p)$ we can derive an obligation $(l_2, O(r), q)$, which reduces this example to the first example given above. This suggests that to detect normative conflicts we need a derivation mechanism, which is a non-trivial problem in itself.

Even when we have a mechanism at our disposal by which we can make derivations from a set of norm instances, there are still situations in which a violation is inevitable, but are nevertheless hard to detect. Consider, for example, an obligation $(l_1, O(p), q)$ and a prohibition $(l_2, F(\text{not } q), \text{not } p)$. The only way to avoid a violation is to satisfy both p and q at the same time. The fact that a violation can be avoided, means that these two norm instances are not in conflict. Now assume that these two norm instances are directed to a single person, p stands for "being in room 1" and q for "being in room 2" with room 1 and room 2 being physically separated rooms. Then a violation is inevitable, because it is physically impossible to be in two rooms at the same time. The problem here is that to conclude that these two norm instances raise a conflict we need background knowledge about the brute state telling us that $\neg(p \wedge q)$. However, such background knowledge is usually not available, at least not in our framework.

Another issue that is closely related to normative conflicts pertains to the coherency of norm schemes (see also (Hansen et al., 2007)). That is, the issue of whether a set of norm schemes generates situations in which a violation is inevitable. If we classify two norm instances $(l_1, F(p), q)$ and $(l_2, O(p), q)$ as conflicting, we should also regard two norm schemes $l_1 : \langle c, F(p), q\rangle$ and $l_2 : \langle c, O(p), q\rangle$ as incoherent. Here also the issue of background knowledge plays an important role. Suppose norm schemes l_1 and l_2 had different conditions, say c and c'. At first sight, these norm schemes seem coherent. But they are incoherent if we know that c' is always true whenever c is true, i.e. $c \rightarrow c'$. Moreover, norm schemes are not only incoherent when they give rise to conflicting norm instances. Consider, for example, a norm scheme $l_3 : \langle c, F(c), q\rangle$. Triggering this norm scheme leads to a norm instance that in itself does not necessarily lead to a normative conflict. But knowing that the forbidden state is already established when the norm instance is asserted, triggering the norm scheme nevertheless inevitably causes a violation.

Above discussion shows the intricacies of defining the concept of normative conflict and normative coherence. Detecting, resolving or even preventing normative conflicts at runtime or design time is not trivial. We strongly feel that model checking techniques might prove useful in verifying and detecting situations in which violations cannot be avoided given a normative specification. This issue is left for future research.

4.6 Conclusion

In this chapter we enriched our organizational artifacts with a normative dimension. We introduced the syntax of norm schemes pertaining to conditional obligations and prohibitions that are accompanied by a deadline. These norm schemes instantiate (unconditional) obligations and prohibitions when their condition is satisfied. We underpinned the syntax with an operational semantics (Plotkin, 1981), which allowed us to formally investigate some essential properties our obligations and prohibitions exhibit. Where norms in existing frameworks typically take on the form of "ought-to-do" statements pertaining to actions that should (or should not) be performed, our norms take on the form of "ought-to-be" statements pertaining to the declarative description of a state that should be achieved (or avoided). To motivate agents to abide by the norms we introduced a sanctioning mechanism to reward obedience and punish infringements.

Chapter 5

Programming Organizational Structure

In chapter 3 we explained the role of organizational artifact in the development of multi-agent systems in which agents that are possibly unknown to the artifacts' designers may dynamically enter and exit. In chapter 4 we further extended our organizational artifacts with a normative component in order to guarantee the global objectives of the artifact (and the system as a whole) to be achieved or maintained. Similar to many existing normative frameworks we used a simple representation of roles in which roles were labels used for, e.g. associating norms (such as obligations and prohibitions) with agents based on the roles they play. This allows us to abstract away from the individuals that will play a role, which is particularly useful in the development of artifacts for which it is not a priori known which agents will interact with them (not to mention the case in which, for example, a secretary gets replaced by another one). This property alone already makes that roles can be considered the cornerstones in constructing open, organization-oriented agent systems.

Indeed, as already shown in chapter 2, roles are a recurring concept in agent design methodologies (see for example (Odell et al., 2003), (Wooldridge et al., 2000), (Zambonelli et al., 2001a) and (Dignum, 2004)) and they are first-class abstractions in several approaches to implementing software systems – within the area of multi-agent systems (see for example (Baldoni et al., 2008), (Baldoni et al., 2009), (Dastani et al., 2004b), (Esteva et al., 2004) and (Hübner et al., 2007)) and outside this area (cf. (Kristensen, 1995; Sandhu et al., 1996; Baldoni et al., 2005, 2007)). The host of different interpretations of the concept of role witnesses the lack of consensus on a crisp definition. Despite little agreement on a definition of role still some properties can be identified we believe should be exhibited by a notion of role for efficiently constructing organization-oriented multi-agent systems:

- Firstly, as argued by Colman and Han (2007) for developing adaptable organizations roles need to be *organization-centric* rather than player-centric. This means that a role exists inside an organization as is the case with the approaches of (Baldoni et al., 2005) and (Baldoni et al., 2008), instead of being an adjunct of its player, an approach taken by, for example, (Kristensen, 1995) and (Dastani et al., 2004b). The organization-centric view promotes the principle of separation of concerns. That is, the organization and its roles can be developed apart from the agents that will play them. Moreover, taking a stance in which agents internalize their roles implies making assumptions about the internal structure of the agents playing the roles.

- Secondly, we observe that roles seem to have a *prescriptive* character. That is to say, they prescribe a behavior to their players and in this way guide the players in how to interact with the organization in a meaningful way. This property is in some way exhibited by all of the previous mentioned approaches. In most approaches by associating norms to roles, and in powerJADE (Baldoni et al., 2008, 2009) and powerJava (Baldoni et al., 2005, 2007), for example, by requiring the role's players to posses certain properties. The prescriptive character of roles also directly follows, for instance, from (a part of) the definition of Odell *et al.* (Odell et al., 2003) who introduce a role as "a class that defines a normative behavioral repertoire of an agent."

- Thirdly, roles are *employable*, meaning that a player can employ its role by "delegating" it a task without the need to know how this task is being executed. This way a role provides a player extra functionality, which might be an incentive for agents to enact roles in the first place. Moreover, in combination with the organization-centric view, this property hides the player from details about how the organization is structured. The roles of (Baldoni et al., 2005), (Baldoni et al., 2008), (Baldoni et al., 2009), (Dastani et al., 2003), (Dastani et al., 2004b), (Kristensen, 1995) all provide functionality that can be used by their players.

- Lastly, we argue that roles should be *BDI-oriented*. BDI-oriented concepts are declarative in nature, and this accords with the declarative nature of other concepts such as norms, obligations and prohibitions by which organizations are specified. Moreover, most agent-oriented programming languages are based on BDI-oriented concepts such as beliefs, goals and planning rules that are equiped with a well-defined formal semantics. By recycling these concepts we rely on a great body of research and it allows the programmer to specify roles in terms he/she is already familiar with. Specifically, assumed that roles are employable, attributing goals to roles seems to offer the right level of abstraction, because agents that interact with the organization then need not be concerned with which low-level procedures to use to reach a certain state. The approaches of, for example, (Dastani et al., 2003, 2004b; Esteva et al., 2004; Sandhu et al., 1996; Hübner et al., 2007) adopt a declarative view on roles, but only Dastani *et al.* (2003; 2004b) explicitly attribute mental attitudes to roles in the conext of agent programming. Also Boella and Van der Torre (2004) have advocated the use of BDI concepts for roles.

Roles are the cornerstones in defining the organizational structure. By inter-role relationships it is, for example, expressed which roles cannot be played by the same agent and which role players an agent may communicate with on the basis of the role it plays. In many existing frameworks that have support for roles and their relational constraints (cf. (Esteva et al., 2004; Hübner et al., 2007; Ferber et al., 2003)), the inter-role relationships can typically be characterized by two properties, that is: 1) they are specified at design time and will unconditionally hold at run-time, and 2) they are seen as hard-constraints. The first property implies that, if it is specified that, say, an agent cannot play the role of reviewer and chair at the same time, then this restriction remains into effect during the whole computation independent of the situation the organization is in. It is in general not possible to express conditional restrictions, for example, that a reviewer cannot be chair given that there are enough reviewers. This also means that above mentioned approaches lack expressiveness to specify more temporal structural constraints such as that an agent should enact a role after another role has been enacted (see for example (Zambonelli et al., 2001a)). The second property implies regimentation of the organizational structure, i.e. agents may not deviate from the prescribed organizational structure. This property undermines the flexibility of the organizational artifact and limits the agents' autonomy (Aldewereld, 2007; Castelfranchi, 2004).

In this chapter we develop a solution for role-based programming constructs aiming to overcome the issues discussed above. In particular, the contributions of this chapter are as follows:

1. We discuss how we extend our notion of organizational artifact (section 5.1) to include roles that exhibit all four key properties identified above. That is to say, roles in our framework are organization-centric, prescriptive, employable and BDI-oriented. We are not aware of any existing organizational frameworks that include roles exhibiting all four key properties.

2. We present programming constructs for implementing roles in section 5.2 and explain them by our running example involving the conference management system. We underpin our syntax by an operational semantics (Plotkin, 1981) in section 5.3. This allows us to explain the meaning of the programming constructs in a mathematically rigourous manner, without needing to commit to a specific implementation platform. Another benefit is that we

bring organization-oriented programming closer to agent-oriented programming languages which are often investigated by an operational semantics.

3. We explore the possibility to express structural constraints for expressing the structure of an organization by means of the normative concepts presented in chapter 4. We demonstrate that the limitations of existing frameworks as discussed above can be overcome (section 5.4). The idea of using norms to express structural constraints has also been suggested in our earlier work (Tinnemeier et al., 2009a) and in a recent work by Hübner *et al.* (2010).

We end this chapter by discussing to what extend the properties that are identified by Steimann (2000) are present in our solution to programming roles and giving directions for future research.

5.1 Organizational Artifacts with Roles

In chapter 3 we explained that in our view a multi-agent system consists of a set of heterogeneous agents that may exploit the functionality provided by organizational artifacts to achieve their goals. An organizational artifact implements the non-autonomous functionalities of a multi-agent system that are better implemented by non-agent concepts. An artifact provides actions that agents may use to manipulate the domain specific state, which is modeled by a set of brute facts. In previous chapter (chapter 4) we enriched the artifact with a normative dimension (i.e. norms and sanctions) to protect the artifact from ending up in sub-ideal states. In this chapter we further extend the notion of an organizational artifact with roles and positions that agents can play. An overview of the internals of an organizational artifact is shown in figure 5.1. In this chapter we will focus on the roles and positions through which agents interact with an artifact.

To interact with the organizational artifact in a meaningful way agents enact roles. Inspired by (Odell et al., 2003) we refer to a role enactment by the term *position*: "a role assignment that can be occupied by at most one agent at a time." An agent can play multiple roles at the same time, and can even play the same role more than once via multiple positions, e.g. being a reviewer for two conference sub-tracks. Positions are computational entities with their own state and behavior, and act as organizational representatives of the agents playing them. Positions are an integral part of the organization, which accords with organization-centric view as put forward by Colman *et al.* (2007). The idea is that agents now interact via their positions, which can be seen as their organizational assistents, guiding them to interact with the artifact in a meaningful way. Similar to BDI agents a position has a mental attitude comprised of beliefs, goals and plans. A player can inspect and configure its position by testing and updating the position's beliefs and goals. By means of its plans a position is able to modify its own internal state, alter the brute state of the organizational artifact (by performing external actions) and communicate with other positions within the

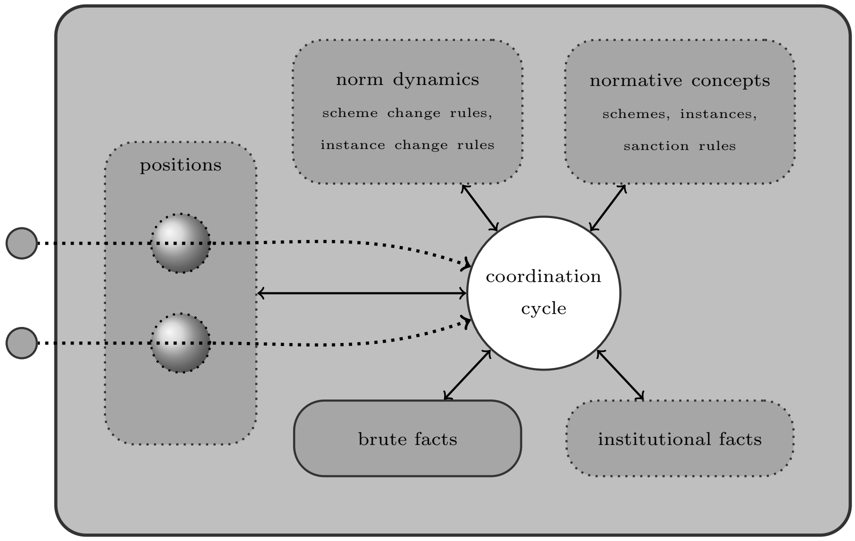


Figure 5.1: A (simplified) conceptual representation of the internals of an organizational artifact. Optional concepts (e.g. norms, positions) are outlined with a dashed border. Solid arrows denote the reading and modification of the concepts as explained in more detail below. Dotted arrows denote actions and messages between agent and artifact. A brief description of all the concepts can be found in chapter 3.

system and send communication messages to its player. In what follows, we will discuss all of these aspects in more detail.

5.1.1 Role Enactment

Before an agent can start interacting with its position, the agent should first notify the organizational artifact that it wishes to enact a role. If the request for enacment is granted by the organization, a position will be instantiated from the role specification. The relation between role and position can thus be compared with the relation between class and object from object-oriented programming. The same roles enacted by different agents share a common type of mental state, yet this state may differ in particular details. For example, the reviewer role has goals and beliefs about papers to review, but which papers and how many is specific to the role enactment at hand. Upon enacting a role these details (some of which may be provided by the enacting agent) are incarnated resulting in an instantiation of the role, i.e. a position. Instantiating a role amounts to executing the role's constructor plan; a special plan for configuring the position's initial state. Likewise, positions may have destructor plans that specify the actions that should be taken upon de-enactment, e.g. de-registering a reviewer from the system. Both constructor and destructors are guarded by a condition specifying the circumstances under which a role may be enacted or a position may be de-enacted. The part of the coordination cycle that explains the enactment and de-enactment of roles is depicted by figure 5.2. Recall that in these pictures, arrows from process (depicted as circle) to store (depicted as rounded box) denote the modification of the store's elements, whereas an arrow in the opposite direction denotes the reading of the store's elements. Observe from 5.2 that a constructor/destrcutor plan is able to perform external actions to modify the artifact's brute state.

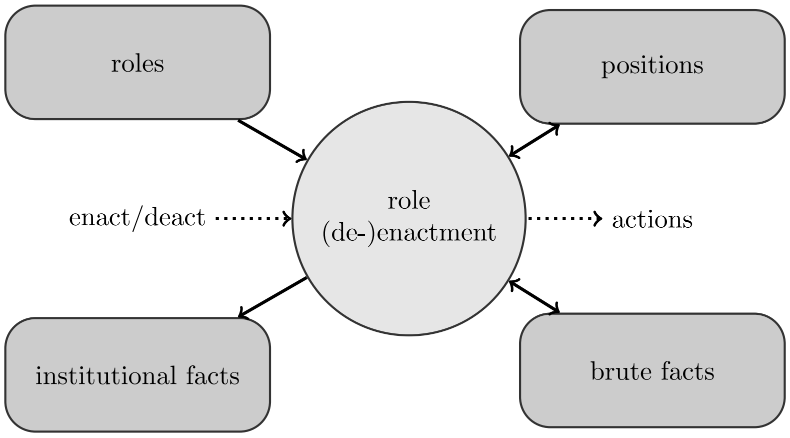


Figure 5.2: Role enactment/de-enactment results in the creation/deletion of a position (a role instance). To initialize/finalize a position is to execute the role's constructor/destructor, i.e. a special plan that may perform actions to alter the position's mental state and the brute state of the organizational artifact. Information about which positions are enacted by which agents is stored by the institutional state.

5.1.2 Position Configuration and Execution

Once enacted, a player can perform special designated actions for configuring its position by inspecting and altering its mental state, in particular, by querying and updating the position's beliefs and goals. A player of the reviewer position can, for example, delegate it a goal to guide its player while reviewing its papers and update the position's beliefs with the written reviews. By activating its position a player can start *employing* its position; similar to (Dastani et al., 2004b) a position can be either active or inactive. An inactive position will not execute any of its plans, whereas an active position will start adopting and executing plans to achieve its (player's) goals. It is important to note that a position obtains its computational meaning by the interaction of its player, and therefore differently from (Odell et al., 2003) in our approach a position cannot exist without an agent associated to it. Indeed, this property explains an important difference between an agent and a position - agents act because they want to, whereas positions act because their players want them to. In other words, positions are not autonomous! The process of the configuration of a position is explained by figure 5.3.

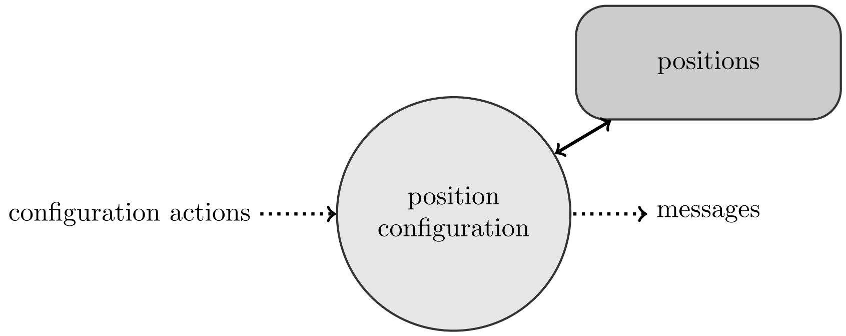


Figure 5.3: Players can configure their positions by performing actions to: 1) inspect the mental state of the position (e.g. testing if certain beliefs hold), 2) manipulate the mental state of the position (e.g. adding/dropping a goal), and 3) activate and de-activate the position. Results of inspection are communicated to their players by means of communication messages.

The actions a position can use in its plans are not limited to actions to modify the positions' own mental state. In fact, positions may act upon the brute state by means of external actions, e.g. for storing a review in a database that can be read by an author position later on. An important characteristic of our approach is that only positions can access and alter the brute facts by performing external actions. That is to say, external agents can only influence the brute facts indirectly by manipulating the mental state of the positions they enact, rather than modifying the brute state directly. Note that this way positions implement a mechanism comparable to Role Based Access Control (RBAC) (Sandhu et al., 1996). Letting only positions act upon the organization's internals promotes the principle of 'data hiding', i.e. the brute facts are encapsulated by the organizational artifact and can only be (partly) read and manipulated by the agents via their positions. This way players are not required to have in-depth knowledge about the artifact's internal state.

Besides performing actions to modify its own mental state and alter the brute facts, a position has some communication abilities. An active position can communicate with other positions within the organizational artifact and can even send its player (but no other agents) communication messages. The reviewer position can, for example, inform its player about its assigned papers once the chair position has informed it about which papers it has been assigned. By allowing a position to communicate with its player, the property of *descriptiveness* is implemented; the position by sending messages can inform its player about what it is expected to do to achieve a certain goal. The advantage of letting positions communicate with each other is that this way many details of the communication structure can be hidden from the players, which we refer to by the term 'communication hiding'. Proper communication protocols can be implemented by the positions and the agents playing them do not need to know how to carry them out and can rely upon their correctness. Moreover, by letting communication flow through the organization more control can be exerted over the communication between agents, facilitating the artifact's ability to regulate the agents' interactions.

The part of the artifact's coordination cycle pertaining to the execution of active positions together with the concepts involved are explained by figure 5.4.

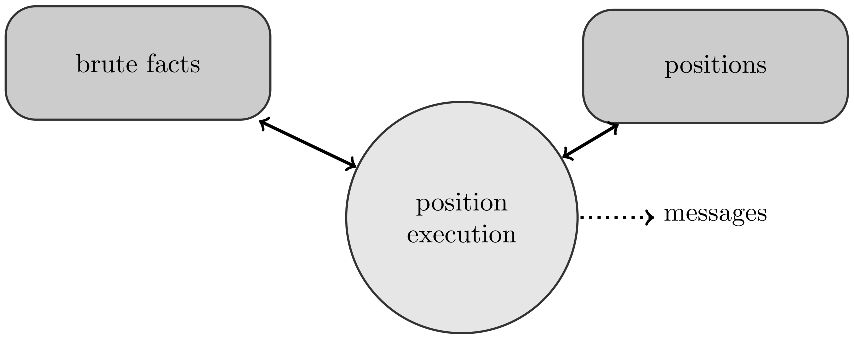


Figure 5.4: Positions execute actions to modify their own mental state, send messages to their players and other positions in the system and manipulate the artifact's brute state.

The 'communication hiding' and 'data hiding' obtained by positions as discussed above has some implications for the functionality of an artifact as proposed up to now. Bear to mind that in chapter 3 we introduced a mechanism for letting the performance of external actions have side-effects, viz. communications messages to be sent to agents interacting with the artifact and external actions to be performed upon other artifacts. Communicating to agents interacting with the artifact is now done via the positions, which makes the ability of communication by the actions' side-effects superfluous. But most importantly, letting artifacts directly act upon each other seems not unifiable with letting interaction with the artifacts only proceed through positions. Unifying both approaches would break the principle of data hiding. Therefore, we decided to exclude the functionality of side-effects in the present chapter. Consequently, it is not possible to link artifacts as we have demonstrated in chapter 3. In section 5.5, in which we give some prospects on future research, we discuss a way to compose artifacts via their positions.

5.1.3 Positions and The Normative Dimension

Although a position is ideally implemented to obey the organizational rules, we still consider a separate monitoring and sanctioning mechanism crucial for the coordination of the overall system. Why norms and sanctions are needed is explained by the synergy of position and player – the behavior of position and player can only be understood in terms of their interaction – and the fact that disallowing violations of the norms drastically decreases the players' autonomy (Grossi, 2007). Why a separate mechanism is needed is explained by three observations:

1. Violations of the rules cannot be fully prevented by the position alone. The position cannot by itself, for example, prevent a violation of the rule that reviews should be uploaded before the notification phase starts.

2. Some rules are hard to implement in the positions because some pertain to the interplay between different position/player pairs, i.e. violations of one or more players might be revealed by actions performed on behalf of another. When the program chair, for example, puts the system in the notification phase this will lead to a violation by all the reviewers which have not yet uploaded their reviews.

3. Finally, and most importantly, the implementation of norms and sanctions is a different concern than the implementation of roles. Separating their implementation increases the manageability and reusability of both concepts.

Therefore, the solution developed in this chapter allows for a seamless integration with the normative constructs proposed in chapter 4. In fact, in section 5.4 we will show how norms can be used for expressing structural constraints, such as incompatibility between roles and cardinality constraints.

Because agents now interact with the artifact via the positions they occupy, we also need information about through which position an agent has enacted a role. For this purpose we introduce another variant of the rea facts, viz. $rea(i, r, p)$ to denote that agent i has enacted role r through a position identified by p. This way we have still access to the agent playing a role, which will prove useful in specifying the norms and sanctions of an artifact. For example, in sanctioning a reviewer for violating its obligation to upload its review in time (see code fragment 4.1 of chapter 4), we still want to blacklist the agent playing the reviewer role. Having access to the positions through which an agent plays its role, will prove useful in programming the roles as we will see in a moment.

5.2 Programming Organizational Artifacts with Roles

Recall that to program an organizational artifact is to specify a set of facts specifying the initial brute state and a set of effects specifying how the brute state

⟨role⟩	a term identifying a role that can be played.
⟨b−atom⟩	a first-order atom denoting a brute fact. The special facts starting with predicate symbol *viol*, *obey* and *rea* (their meaning to be explained later on) are excluded from the set of brute facts.
⟨r − atom⟩	a first-order atom of the form $rea(i, r)$ or $rea(i, r, p)$ in which r denotes a role and i the agent playing it and p the identifier of the position through which the role is enacted; "rea" is short for role enacting agent.
⟨i − atom⟩	a first-order atom of the form $viol(\phi_l)$ or $obey(\phi_l)$ with ϕ_l a ⟨label⟩ identifying a norm. The first denotes a violation of norm labeled ϕ_l whereas the latter denotes the obedience to the norm.

Table 5.1: Elementary syntactical constructs.

evolves under the performance of actions. Optionally, normative concepts such as norm schemes and sanctioning rules may be put into place to regulate the behavior of the artifact's unknown participants as shown in chapter 4. The grammar of the basic constructs can be found in figure 3.6 of chapter 3 and that of the normative components in figure 4.5 chapter 4. All these constructs were explained in detail in previous chapters and will not be repeated here.

In this section we replace our simple view on roles, in which roles are merely labels, by a richer account of roles that exhibit the four properties as explained above. We describe the syntax and intuitive semantics of the programming constructs by which this new notion of roles of an organizational artifact can be programmed. Again, we use the conference management system example to show how roles can be programmed and to explain their intuitive semantics. The meaning of some elementary syntactical constructs that were introduced in previous chapters are repeated in table 5.1.

As discussed in previous section, we restrict the artifact's ability to let the performance of actions have side-effects. To do so, we slightly change the definition of the syntax of the ⟨post⟩ clause (originally defined in figure 3.6 of chapter 3) to no longer contain the special atoms *Send* and *Do*, that are responsible for causing side-effects. We used the ⟨post⟩ clause in defining the post-condition of effect specifications in figure 3.6 of chapter 3. In concrete, we replace the previously defined clause ⟨post⟩ by the following clause[1]:

$$\langle\text{post}\rangle = \langle\text{b} - \text{lit}\rangle \;\mid\; \langle\text{post}\rangle \; \{ \; \texttt{","} \; \langle\text{post}\rangle \; \};$$

For the specification of the syntax of roles the first-order atom with predicate name `construct` is denoted by ⟨constr_atom⟩ and the first-order atom with predicate name `destruct` by ⟨destr_atom⟩. Moreover, by ⟨pc_atom⟩ we denote first-order atoms of the form `msg`(s, p, c) with s the sender of the message, p the performative and c the content of the message. To introduce our new notion of

[1]The same restriction applies to the sanctioning rules presented in chapter 4, but this is not shown here.

```
⟨Role⟩          = ⟨role⟩ "= {" [ ⟨beliefs⟩ ] [ ⟨goals⟩ ] ⟨constructors⟩ [ ⟨pgrules⟩ ]
                  [ ⟨pcrules⟩ ] ⟨destructors⟩ "}";
⟨beliefs⟩       = "Beliefs:" ⟨belief⟩ { ⟨belief⟩ };
⟨goals⟩         = "Goals:" ⟨goal⟩ { ⟨goal⟩ };
⟨constructors⟩  = "Constructors:" ⟨constructor⟩ { ⟨constructor⟩ };
⟨pgrules⟩       = "PG-rules:" ⟨pgrule⟩ { ⟨pgrule⟩ };
⟨pcrules⟩       = "PC-rules:" ⟨pcrule⟩ { ⟨pcrule⟩ };
⟨destructors⟩   = "Destructors:" ⟨destructor⟩ { ⟨destructor⟩ };
⟨constructor⟩   = ⟨constr_atom⟩ "<-" ⟨test⟩ "|" ⟨plan⟩;
⟨pgrule⟩        = ⟨goalquery⟩ "<-" ⟨belquery⟩ "|" ⟨plan⟩;
⟨pcrule⟩        = ⟨pc_atom⟩ "<-" ⟨belquery⟩ "|" ⟨plan⟩;
⟨destructor⟩    = ⟨destr_atom⟩ "<-" ⟨test⟩ "|" ⟨plan⟩;
⟨test⟩          = "B(" ⟨belquery⟩ ")" | "G(" ⟨goalquery⟩ ")" |
                  | "E(" ⟨envquery⟩ ")" | "I(" ⟨instquery⟩ ")" | ⟨test⟩ "&" ⟨test⟩;
⟨envquery⟩      = "true" | ⟨b − lit⟩ | ⟨envquery⟩ "and" ⟨envquery⟩
                  | ⟨envquery⟩ "or" ⟨envquery⟩;
⟨instquery⟩     = "true" | ⟨i − lit⟩ | ⟨instquery⟩ "and" ⟨instquery⟩
                  | ⟨instquery⟩ "or" ⟨instquery⟩;
⟨b − lit⟩       = "true" | ⟨b − atom⟩ | "not" ⟨b − atom⟩;
⟨i − lit⟩       = "true" | ⟨r − atom⟩ | ⟨i − atom⟩ | "not" ⟨r − atom⟩ |
                  | "not" ⟨i − atom⟩;
```

Figure 5.5: EBNF grammar of the minimal syntax constituting a role.

roles we replace the previously defined clause ⟨roles⟩ as defined in figure 3.6 of chapter 3 by the following clause:

⟨roles⟩ = "Roles:" ⟨Role⟩ { ⟨Role⟩ };

in which the syntax for specifying roles (i.e. the element ⟨Role⟩) is defined as depicted in figure 5.5. The syntacticial element ⟨role⟩ is as specified before, i.e. a label uniquely identifying the name of a role that can be played. Note that as for now the full role specifications are embedded in the code defining the organizational artifact. We envisage that in an actual implementation the code that specifies the role is located in a separate file.

For the sake of generality we leave the language by which roles are implemented largely unspecified. That is to say, we deliberately leave the specification of some syntactical elements such as beliefs, goals, the queries that can be performed on them, and plans open. We do so, to encourage the reuse of (parts of) existing BDI-based programming languages endowed with a formal semantics, for example GOAL (de Boer et al., 2007), AgentSpeak(L) (Rao, 1996), Jason (Bordini et al., 2005) or 2APL (Dastani, 2008). Figure 5.5 includes what we believe to be the minimal set of programming constructs a role should have. Additionally, we assume the plans (denoted by ⟨plan⟩) to at least include actions for acting upon the brute facts and actions for communicating with its player and other positions present in the system. It should be noted, however, that also the minimal syntax of the elements presented above might vary depending on the specific agent-oriented programming language at hand.

Each of the core ingredients of the programming language by which roles can be programmed will be explained below. For the sake of illustration we assume that the account of roles presented here is based on the 2APL programming language (Dastani, 2008) (which also explains why the syntax of roles closely resembles the one of 2APL). A brief overview of the 2APL language can be found in chapter 2. We explain the syntactical elements involved in programming roles by means of a simplified, partial implementation of a reviewer role for our conference management system example. The example is by no means meant to be exhaustive; we merely use it to show how roles may be implemented.

As explained before, a position is an instantiated role encapsulating the state (e.g. beliefs and goals) of a specific role enactment. When enacting a role a position containing the initial details about the role enactment is instantiated. Inspired by object-oriented programming we introduce the notion of a constructor, a plan that is executed upon instantiation with the responsibility to put a freshly created position in a valid initial state. A constructor is a rule of the form `construct(`$t_1, ..., t_n$`) <-` Φ `|` π. The terms $t_1, ..., t_n$ of atom *construct* are the arguments that are provided by the role enactant with which it wishes to enact the role. It is assumed that the first parameter is always the identifier of the agent enacting the role. In many cases a role can only be enacted under certain circumstances. The reviewer role can, for example, only be enacted by players that are invited. For this reason a constructor plan is guarded by a pre-condition Φ. This pre-condition is a query that is evaluated over the beliefs and goals of the position (marked by B and G), the institutional state (marked by I) and the brute state (marked by E from environment). Finally, the sequence of actions π specifies the actions that should be taken to put the position in its initial state. Once the whole constructor plan has completed its execution the position is added to the organizational artifact and the player is sent its unique identifier in order to interact with it. While the constructor plan π is still executing the position is not yet properly instantiated. Interacting with a position that is not yet in a valid state might lead to erroneous results. A position should thus not publish its identity while still in the process of being constructed. This suggests that communication actions cannot be used in a role's constructor.

The constructors of the reviewer role are displayed in code fragment 5.1 on lines 15 – 27. The first constructor states that a player is allowed to enact the reviewer role during the submission phase and when invited (which is stored by brute facts). To initialize this position is then to store the identity of the player as a belief by means of the belief update specified on line 7 and, similarly, to store the identity of the chair. Note that the identity of the chair is obtained by querying the institutional state in the role's pre-condition. The second constructor takes a second argument by which the player may indicate a maximum number of papers it wishes to review. The constructor's pre-condition ensures that a reviewer agrees

Code fragment 5.1 Implementation of a reviewer role.

```
reviewer =                                                                 1
{                                                                          2
Beliefs:                                                                   3
  maxReviews(5).                                                           4
                                                                           5
BeliefUpdates:                                                             6
  { } SetPlayer(P) {player(P)}                                             7
  { } SetChair(C) {chair(C)}                                               8
  {maxReviews(X)} SetMaxReviews(Y) {not maxReviews(X), maxReviews(Y)}      9
  {not told(X)} Tell(X) {told(X)}                                          10
  { } Assigned(PaperId) {assigned(PaperId)}                                11
  { } UploadReview(PaperId) {reviewed(PaperId)}                            12
                                                                           13
Constructors:                                                              14
  construct(Player)                                                        15
  <- E(phase(submission) and invited(Player)) & I(rea(C,chair,P)) |        16
  { SetPlayer(Player);                                                     17
    SetChair(P)                                                            18
  }                                                                        19
                                                                           20
  construct(Player, maxReviews(X))                                         21
  <- E(phase(submission) and invited(Player))                              22
     & I(rea(C,chair,P)) & B(X≥3) |                                        23
  { SetPlayer(Player);                                                     24
    SetChair(P);                                                           25
    SetMaxReviews(X)                                                       26
  }                                                                        27
                                                                           28
PG-rules:                                                                  29
  true                                                                     30
  <- not told(maxReviews) and maxReviews(X) |                              31
  { send(C, inform, maxReviews(X) );                                       32
    Tell(maxReviews)                                                       33
  }                                                                        34
                                                                           35
  reviewed(PaperId)                                                        36
  <- decision(PaperId,Decision) and assigned(PaperId) |                    37
  { @org(uploadReview(PaperId, Decision), _);                              38
    UploadReview(PaperId)                                                  39
  }                                                                        40
                                                                           41
PC-rules:                                                                  42
  message(C, inform, assigned(PaperId))                                    43
  <- player(P) and chair(C) |                                              44
  { adopta( reviewed(PaperId) );                                           45
    Assigned(paperId);                                                     46
    send(P, request, reviewed(PaperId) )                                   47
  }                                                                        48
                                                                           49
Destructors:                                                               50
  destruct() <- E(not phase(review)) | { }                                 51
}                                                                          52
```

to review at least three papers. This additional information is stored as a belief by performance of the belief update action specified on line 9. This update will replace the default initial belief that a reviewer will have to review at most five papers as specified on line 4.

During its lifetime a position may communicate with other positions by means of message passing. To be able to handle messages from other positions we append PC-rules (borrowed from 2APL) to a role's repertoire of programming constructs. Recall that 2APL's PC-rules are of the form ϕ `<-` Φ `|` π allowing the agent to adopt a plan π if communication message ϕ is received and belief expression Φ can be entailed by the agent's beliefs. An example of such a PC-rule is shown on lines 43 – 48 of code fragment 5.1. This rule handles a message of the chair informing the reviewer position that it has been assigned a particular paper to review. In reaction to this information the reviewer position automatically adopts a goal to have reviewed this paper, updates its beliefs by adding a belief that it is assigned this paper and sends a messages to its player requesting it to review this paper. It is thus the player who should actually review the paper, who is assumed to inform its position about its decision by adding a belief of the form `decision(PId,Decision)` to the position's belief base.

To reach its (player's) goals a position needs to adopt plans. For this purpose we equip a position with planning-goal rules (PG-rules), which are directly borrowed from 2APL. PG-rules are rules of the form κ `<-` Φ `|` π with the intuitive meaning that plan π can be used for reaching goal κ under circumstances Φ. The PG-rules of the reviewer role are listed on lines 30 – 40 of code fragment 5.1. The first rule is an example of a reactive rule, that states that when it is not believed that the chair has been notified already about the maximum number of papers the reviewer desires to review, the desired number of reviews will be sent to the chair who uses it in assigning papers to reviewers. The player of the reviewer role is oblivious about the communication taking place with the chair, which has the benefit that playing the role of reviewer is simplified from the player's point of view. The second PG-rule states if a goal has been adopted to have reviewed an assigned paper and there is a belief about a decision for that paper, then a plan will be adopted that uploads the decision by performing an external action in the organizational artifact and updates the beliefs with the fact that the review for that paper has been completed.

Similar to enacting a role, a player may often only quit playing its role under certain conditions. Moreover, upon de-enactment the position possibly needs to perform some operations to safely de-enact its role, e.g. informing another position about the de-enactment. Therefore, we introduce destructors for specifying the conditions under which a role may be de-enacted and the final actions that should be performed. More specifically, a destructor is a rule of the form `destruct() <-` Φ `|` π with the intuitive meaning that plan π should be executed upon de-enactment of the position, which can be done under condition Φ (just like the constructor ranging over the position's mental state and artifact's internal state). When de-enacting the reviewer role, no actions need to be performed. The reviewer role can only be de-enacted when not in the reviewing phase. This is listed on line 51 of code fragment 5.1.

5.3 Executing Organizations with Positions

In the previous section we have defined the syntax of the programming constructs by which roles can be programmed. Moreover, we have shown an example of such an implementation. In this section we define the meaning of these programming constructs and the operations that can be performed on positions by their players by means of an operational semantics (Plotkin, 1981).

5.3.1 Preliminaries

In our system roles are a core ingredient in implementing organizational artifacts. As an instance of a role a position has a state that changes during its execution. In particular, this state is composed of beliefs denoting the information the position has about its environment including its player, goals describing the situations to be established by the position, and a set of plans the position has currently adopted for achieving its objectives. Further, a position has an identifier associated to it for uniquely identifying the position and a flag to indicate the position's activation status. Again, we do not make any assumptions about the exact structure of a position's internal mental components for the sake of generality; their structure depends on the choice of the agent-oriented programming language.

Definition 5.1 (Position Configuration) *A position, typically denoted by ρ, is a tuple $\langle p, \Sigma, \Gamma, \Pi, f \rangle$, in which:*

- *p is the label uniquely identifying the position. This name has the form $i.k$ with k a unique identifier and i an agent identifier denoting the player. From here on we say that a position p is played by i iff $p = i.k$;*
- *Σ is a set of beliefs representing the belief base of the position;*
- *Γ is a set of goals representing the goal base of the position;*
- *Π is a set of adopted plans generated by the PG-rules;*
- *$f \in \{\top, \bot\}$ is a boolean flag indicating the activation status of the position, $\top$ denotes active and $\bot$ inactive.*

In the following we say that a position $\langle p, \Sigma, \Gamma, \Pi, f \rangle$ with $f = \top$ is active.

To explain the execution of positions and the interaction with their players, we further extend our notion of organizational artifact as originally introduced in chapter 3.

Definition 5.2 (Artifact Configuration) *An organizational artifact, typically denoted by O, is defined as a tuple $\langle id, \mathrm{Roles}, \varrho, \sigma_b, \sigma_i, \mathrm{E}, \epsilon_{in}, \epsilon_{out} \rangle$, in which:*

- *id is a name uniquely identifying the artifact;*

- Roles *is a set of tuples* (r, s) *in which* s *is the program specification of the role uniquely identified by label* r*.*
- ϱ *is a set of positions (role instances);*
- σ_b *is a set of ground first-order atoms describing the brute state of the artifact;*
- σ_i *is a set of ground first-order atoms describing the artifact's institutional state;*
- E *is a set of effect specifications;*
- ϵ_{in} *is a list of ground first-order atoms, the events perceived by the artifact;*
- ϵ_{out} *is a list of ground first-order atoms with predicate name Send, the communication messages to be sent to agents interacting with the organizational artifact.*

Optionally, the normative components that were introduced in the previous chapter could be included in the definition of the organizational artifact. Because the transition rules specifying their meaning are defined independently from the transition rules for the extension with roles introduced in present chapter, we decided not to include them here.

As before, we define the notion of an initial artifact configuration, redefining the one previously defined in chapter 3. The initial organizational artifact is the one that is determined by the program code. The artifact's program defines the artifact's name, its initial brute state, the effect specifications, and the roles that can be played. Initially, no events are received, no events need to be sent, no roles have been instantiated, and no roles have been enacted. This is defined by the following definition.

Definition 5.3 (Initial Artifact Configuration) *An organizational artifact configuration of the form* $\langle id, \text{Roles}, \emptyset, \sigma_b, \emptyset, \text{E}, \emptyset, \emptyset \rangle$ *specified by a program such that id is specified by the program's name component,* Roles *is specified by the program's roles component,* σ_b *is characterised by the program's facts component, and* E *is defined by the program's effect component, is called an initial artifact configuration.*

At the beginning of a computation of an organizational artifact, no positions are thus instantiated. Positions are instantiated when an agent requests to enact a role and that request matches with a constructor of that role, i.e. the parameters provided by the agent match and the precondition of the role is fulfilled. As we have seen in the example discussed in previous section, the condition of a constructor (and destructor) allow a test on its intrinsic state (goals and beliefs) and extrinsic state (brute and institutional facts). Hence the definition of the special entailment operator $\models_t$ that evaluates such a test.

Definition 5.4 *Given* ⟨test⟩ *expressions* $\varphi(\overline{x})$ *and* $\varphi'(\overline{y})$ *in which sets of free variables* $\overline{x}$ *and* $\overline{y}$ *occur,* ⟨belquery⟩ *expression* ϕ_b*,* ⟨goalquery⟩ *expression* ϕ_g*,* ⟨envquery⟩ *expression* ϕ_e*,* ⟨instquery⟩ *expression* ϕ_i*, first-order entailment relation* $\models$ *and substitution function* θ*, the entailment relation* $\models_t$ *that evaluates test expressions w.r.t. beliefs, goals, brute facts and institutional facts* $(\Sigma, \Gamma, \sigma_b, \sigma_i)$ *is defined as:*

- $(\Sigma, \Gamma, \sigma_b, \sigma_i) \models_t \mathtt{B}(\phi_b)\theta$ iff $\Sigma \models \phi_b\theta$
- $(\Sigma, \Gamma, \sigma_b, \sigma_i) \models_t \mathtt{G}(\phi_g)\theta$ iff $\exists \gamma \in \Gamma$ s.t. $\gamma \models \phi_g\theta$
- $(\Sigma, \Gamma, \sigma_b, \sigma_i) \models_t \mathtt{E}(\phi_e)\theta$ iff $\sigma_b \models \phi_e\theta$
- $(\Sigma, \Gamma, \sigma_b, \sigma_i) \models_t \mathtt{I}(\phi_i)\theta$ iff $\sigma_i \models \phi_i\theta$
- $(\Sigma, \Gamma, B, I) \models (\varphi(\overline{x})\&\varphi'(\overline{y}))\theta$ iff $\exists\theta_1 : [\theta_1 = \theta|\overline{x}$ and $(\Sigma, \Gamma, \sigma_b, \sigma_i) \models \varphi\theta_1$ and $\exists\theta_2 : [\theta_2 = \theta|(\overline{y} \setminus \overline{x})$ and $(\Sigma, \Gamma, \sigma_b, \sigma_i) \models \varphi'\theta_1\theta_2]]$
  with '$|$*' to be read as 'restricted to the domain'.*

Note that we make some assumptions about the structure of the belief and goal base. In accordance with most agent-oriented programming languages we assume that both goal base and belief base are based on some subset of first-order logic. We need to make such assumptions because we introduce actions to change the positions' goals and beliefs. Also note that the derivation of goals is different from the entailment of beliefs. It is such that from a goal base $\Gamma = \{\phi_1 \wedge \phi_2, \phi_3\}$ we can derive, for example, ϕ_1, but not $\phi_1 \wedge \phi_3$. Other possible goal entailment relations can be found in (van Riemsdijk et al., 2005).

Recall that we distinguish between transitions at the multi-agent system level consisting of agents and organizational artifacts (denoted by an arrow $\longrightarrow_{mas}$), transitions made by agents (denoted by $\longrightarrow_{agt}$), and transitions of the organizational artifact (denoted by $\longrightarrow_{org}$). Because positions are computational entities with a state that may change during execution, we also introduce transitions to refer to the execution steps made by a position. We denote these transitions by an arrow $\longrightarrow_{pos}$. In defining these compositional transition systems we use a bottom-up approach. That is to say, we start with defining the transitions agents and positions may make. Then we define the transition system from which the transitions of the organizational artifact can be derived. The transitions at the multi-agent level that explain the interaction between agents and organizational artifacts are defined in chapter 3. These derivation rules remain the same and will not be presented in this chapter.

5.3.2 Executing Positions

The basic types are the transitions that define the execution of an agent and the transitions that define the execution of a position. The transitions that explain the execution steps an agent may make were explained in chapter 3 and will not be repeated here. Bear to mind, however, that we do treat agents as black-boxes and, consequently, did only list the transitions an agent can make without defining the transition system by which these transitions can be derived. In particular, we assumed that an agent can make internal transitions to change its own mental state, perform observable external actions, and send messages to communicate with other agents.

The programming constructs by which positions can be programmed show many similarities with that of agent programming languages. To preserve generality of our approach we do not commit to a particular agent programming language for programming positions. In fact, the semantics of a position is similar to that of agent programming languages and is defined elsewhere, cf. (Rao, 1996; de Boer et al., 2007; Bordini et al., 2005; Dastani, 2008). Therefore, similar to agents, we do not present the transition system by which transitions for positions can be derived. Without loss of generality we assume that active positions can execute external actions δ (of type $\langle\text{atom}\rangle$) that at the organizational level change the brute state. We also assume that positions can make internal transitions that only affect their internal state. Examples of such internal actions are updates of the belief base and the addition and deletion of goals. Moreover, we assume that positions can send messages to their players and other positions within the same organizational artifact. More precisely, we assume transition rules by which the following transitions for an *active* position ρ can be derived:

1) $\rho \xrightarrow{P(i,t_1,\ldots,t_n)!}_{pos} \rho'$ performance of external action named P with a possibly empty list of arguments $t_1, ..., t_n$;

2) $\rho \xrightarrow{msg(i,j,p,c)!}_{pos} \rho'$ position i sends message to receiver j with performative p and content c;

3) $\rho \longrightarrow_{pos} \rho'$ performance of non-observable internal action.

Additionally, we assume that positions can always receive communication messages. That is, even if the position is not active, it is able to receive the message to be handled when activated again. How these messages are stored and handled by the position depends on the agent-oriented programming language that is chosen.

4) $\rho \xrightarrow{msg(j,i,p,c)?}_{pos} \rho'$ position i receives message from sender j with performative p and content c.

As appears from the transitions above, a position is capable of sending and receiving communication messages. The transition system explaining the execution of organizational artifacts, which we present in next section, will be designed such that the communication of a position is limited to sending and receiving messages to/from its player, and sending and receiving messages to/from other positions within the organizational artifact. In other words, positions cannot communicate with any other agent than their player.

5.3.3 Execution of Organizational Artifacts

In defining the transition system for organizational artifacts we assume an artifact configuration $\langle id, \mathrm{Roles}, \varrho, \sigma_b, \sigma_i, \mathrm{E}, \epsilon_{in}, \epsilon_{out}\rangle$. For the sake of presentation, the components that will not be modified by the transition rules that will be defined below will be omitted. That is, the identifier id, the set of roles Roles and the set of effect specifications E.

Before we explain how agents take up roles we first introduce the transition rules that define the execution of positions, because these transitions will be used in defining position enactment and de-enactment later on. Most actions of a position only affect its internal state and leave the state of the organizational artifact unchanged. Recall that a position can only advance in its computation if it is activated by its player. Internal transitions are defined by the following rule.

Transition Rule *Let $\rho = \langle p, \Sigma, \Gamma, \Pi, \top\rangle$ be an active position, then the rule for internal actions performed by a position is defined as:*

$$\frac{\rho \in \varrho \quad \rho' = \langle p, \Sigma', \Gamma', \Pi', \top\rangle \quad \rho \longrightarrow_{pos} \rho'}{\langle \varrho, \sigma_b, \sigma_i, \epsilon_{in}, \epsilon_{out}\rangle \longrightarrow_{org} \langle (\varrho \setminus \{\rho\}) \cup \{\rho'\}, \sigma_b, \sigma_i, \epsilon_{in}, \epsilon_{out}\rangle} \qquad (psilent)$$

In previous chapters, agents were capable of performing external actions in order to change the artifact's brute facts directly. How the organizational artifact responds to external actions performed by agents is defined by transition rule *effect* of chapter 3. Adopting roles and positions, agents only act upon the artifact's brute state indirectly via the positions they hold. In this chapter we introduce another transition rule for handling external actions performed by some active position. This rule is meant to replace the formerly defined transition rule *effect*. Recall the consistency-preserving update operator $\uplus$ defined by definition 3.3 of chapter 3. Also recall that the entailment operator $\models$ and the unification function *unify* are defined in chapter 3.

Transition Rule *Let α' be an external action. Then the rule for processing an external action by applying an effect specification is defined as follows:*

$$\frac{\rho \in \varrho \quad \rho \xrightarrow{\alpha'!}_{pos} \rho' \quad (\Phi, \alpha, \Psi) \in \mathrm{E} \quad unify(\alpha, \alpha') = \theta_1 \quad \sigma_b \models \Phi\theta_1\theta_2}{\langle \varrho, \sigma_b, \sigma_i, \epsilon_{in}, \epsilon_{out}\rangle \longrightarrow_{org} \langle (\varrho \setminus \{\rho\}) \cup \{\rho'\}, \sigma'_b, \sigma_i, \epsilon_{in}, \epsilon_{out}\rangle} \qquad (effect)$$

where $\sigma'_b = \sigma_b \uplus \{\phi \mid \phi \in \Psi\theta_1\theta_2\}$

During their life-time positions may communicate with other positions. For example, a reviewer position may inform the chair about the maximum number of papers it wishes to review (as shown in code fragment 5.1). In case of inter-position communication no message leaves the organizational artifact, because communication can only take place between positions that are member of the same artifact. So, there is no need to store the message to be delivered in the artifact's list of out events; we rather deliver the message instantaneously. This has the additional advantage that we do not need to account for the situation in which the receiving position is de-enacted before the message could be delivered, which would complicate the semantics. Sending a communication message to another position changes the internal state of the sender and that of the receiver as expressed by transitions 2 and 4 of previous section.

Transition Rule *Let ρ_1 be an active position identified by i and let ρ_2 be a position identified by j. Then the rule for message synchronization between positions is defined as follows:*

$$\frac{\rho_1 \in \varrho \quad \rho_2 \in \varrho \quad \rho_1 \overset{msg(i,j,p,c)!}{\longrightarrow}_{pos} \rho_1' \quad \rho_2 \overset{msg(i,j,p,c)?}{\longrightarrow}_{pos} \rho_2'}{\langle \varrho, \sigma_b, \sigma_i, \epsilon_{in}, \epsilon_{out}\rangle \longrightarrow_{org} \langle (\varrho \setminus \{\rho_1, \rho_2\}) \cup \{\rho_1', \rho_2'\}, \sigma_b, \sigma_i, \epsilon_{in}, \epsilon_{out}\rangle} \qquad (ppmsg)$$

Positions not only communicate with other positions in the system, they also communicate extensively with their player. In case a position sends a communication message to its player, this message does leave the organizational artifact. Therefore, we append it to the list of out events to be processed by the artifact later on. This processing boils down to propagating these messages to the multi-agent level such that their addressee can perceive and transact them, as explained by transition rule *send* defined in chapter 3. Transition rule *send*, however, presupposes that messages are sent by the artifact and is such that messages will have the artifact as sender. As explained before, we deprive the artifact of its capability to communicate; only positions may send communication messages. Therefore, we slightly modify the previously defined rule *send* to deal with messages sent by positions. Using a similar notation as in chapter 3, messages that need to be sent are stored in the list of out events as special atoms of the form $Send(i, r, p, c)$ meaning that position identified by i sends messages with performative p and contents c to receiver r (its player or another position within the artifact).

Transition Rule *The rule for sending messages that may originate from either a position or the artifact is defined as follows:*

$$\frac{}{\langle \sigma_b, \sigma_i, \epsilon_{in}, [Send(i, r, p, c)] : \epsilon_{out}\rangle \overset{msg(i,r,p,c)!}{\longrightarrow}_{org} \langle \sigma_b, \sigma_i, \epsilon_{in}, \epsilon_{out}\rangle} \qquad (send)$$

With this rule we can now define the transition rule pertaining to the communication from a position with its player. Remember that the identifier of the position has the form $i.k$ with k a unique identifier and i the identifier of its player. We use this information to guarantee that a position cannot communicate with any other agents than its player.

Transition Rule *Let $\rho = \langle i.k, \Sigma, \Gamma, \Pi, \top \rangle$ be an active position. Then the rule for sending a message from a position to its player is defined as follows:*

$$\frac{\rho \in \varrho \quad \rho \overset{msg(i.k,i,p,c)!}{\longrightarrow_{pos}} \rho'}{\langle \varrho, \sigma_b, \sigma_i, \epsilon_{in}, \epsilon_{out} \rangle \longrightarrow_{org} \langle (\varrho \setminus \{\rho\}) \cup \{\rho'\}, \sigma_b, \sigma_i, \epsilon_{in}, \epsilon_{out} : [Send(i.k, i, p, c)] \rangle} \quad (pamsg)$$

An agent i indicates that it desires to take up a role r with a list of parameters $t_1, ..., t_n$ by performing an action $enact(i, r, t_1, ..., t_n)$. Recall that actions performed by external agents are stored in the artifact's received events. A role can be enacted only if the organizational rules are respected. More precisely, when the arguments $t_1, ..., t_n$ match with the arguments of some constructor of which the pre-condition is satisfied. If so, a fresh position is instantiated out of the role specification by executing its constructor. Evaluating the constructor's condition and execution of its plan occurs as one computation step, i.e. atomically. This way, while executing the constructor, it is guaranteed that the constructor's pre-condition may only cease to hold due to one of the constructor's actions. Moreover, the programmer is relieved from well-known concurrency issues while putting the position in a valid state. A disadvantage of executing the constructor in one shot, however, is that no normative assessment can take place while executing the constructor.

The execution of the constructor boils down to creating an initial position ρ out of the role specification and putting the constructor's plan in ρ's plan base, and performing all derivation rules until its plan base is empty, i.e. until the constructor plan is fully executed. Because the execution of the constructor's plan base may include external actions to modify the artifact's brute state, its execution can only be explained in the context of the artifact configuration. More specifically, the execution of a position's constructor is denoted by a sequence of transitions:

$$\langle \{\rho\}, \sigma_b, \sigma_i, \epsilon_{in}, \epsilon_{out} \rangle \longrightarrow^*_{org} \langle \{\rho'\}, \sigma'_b, \sigma_i, \epsilon_{in}, \epsilon_{out} \rangle$$

that are to be derived from transition rules *effect* and *psilent*. Recall that according to these rules only active positions can be executed, whereas similar to (Dastani et al., 2004b) an enacted position is initially inactive. This explains the need for de-activating active position ρ'. If the position is successfully instantiated the agent is informed about this and is sent the identifier of the position such that it can interact with it. Moreover, the fact that the agent identified by i has enacted role r through position identified by p is stored as an institutional fact $rea(i, r, p)$. Note that these facts differ from the binary $rea(i, r)$ facts we used up to now. The next transition rule defines role enactment.

Transition Rule *Let* $(r, s) \in$ Roles *such that* `construct(`Ψ_1`) <-` Φ `|` π *is a constructor of role specification s with Ψ_1 a list of terms (the constructor's parameters) and let Ψ_2 be a list of ground terms (the parameters provided by the* *agent through the enact action). Moreover, recall the function unify*(Ψ_1, Ψ_2) *that evaluates to the most general unifier* θ *of* Ψ_1 *with* Ψ_2 *if they are unifiable and returns* $\perp$ *otherwise. Further, let* Σ *be a belief base characterised by the belief component of role specification s and let* Γ *be a goal base specified by the goals of role specification s. Then the rule for role enactment is defined as:*

$$\frac{\begin{array}{c} \mathit{unify}(\Psi_1, \Psi_2) = \theta_1 \quad (\Sigma, \Gamma, \sigma_b, \sigma_i) \models_t \Phi\theta_1\theta_2 \\ \langle \{\rho\}, \sigma_b, \sigma_i, \epsilon_{in}, \epsilon_{out} \rangle \longrightarrow^*_{org} \langle \{\rho'\}, \sigma'_b, \sigma_i, \epsilon_{in}, \epsilon_{out} \rangle \end{array}}{\begin{array}{c} \langle \varrho, \sigma_b, \sigma_i, [\mathit{enact}(i, r, \Psi_2)] : \epsilon_{in}, \epsilon_{out} \rangle \\ \longrightarrow^*_{org} \\ \langle \varrho \cup \{\rho''\}, \sigma'_b, \sigma'_i, \epsilon_{in}, \epsilon_{out} : [\mathit{Send}(p, i, \mathit{inform}, \mathit{enacted}(r, p))] \rangle \end{array}} \qquad (\mathit{enact})$$

$$\begin{array}{lll} \text{where } p & = & i.k \text{ in which } k \text{ is a unique identifier} \\ \rho & = & \langle p, \Sigma, \Gamma, \{\pi\theta_1\theta_2\}, \top \rangle \\ \rho' & = & \langle p, \Sigma', \Gamma', \{\}, \top \rangle \\ \rho'' & = & \langle p, \Sigma', \Gamma', \{\}, \perp \rangle \\ \sigma'_i & = & \sigma_i \cup \{\mathit{rea}(i, r, p)\} \end{array}$$

Enactment of a role might not be successful for many reasons. For example, when the role does not exist in the artifact, the pre-conditions of role enactment are not respected or in case the parameters do not match with any constructor, but also when the plan's constructor somehow fails to execute. The circumstances under which a constructor plan cannot advance in its computation depends on the semantics of the agent-oriented programming language chosen. The definition of the transition rules covering all the cases in which role enactment fails are beyond the scope of this chapter, although we stress that in an implementation a failure handling mechanism notifying the agent when the enactment fails should be present.

Once an agent occupies a position, the agent can start interacting with it. It does so by performing designated actions, e.g. `activate(i,p)` to indicate that agent i wishes to activate position p. The first parameter of an action is always the agent performing it. Because the agent's identity is coded in the identifier p that is of the form $i.k$ with i denoting the position's player, we can guarantee that an agent can only interact with its own position(s). In what follows all the actions an agent can perform to configure a position it plays are explained in more detail.

An agent may change the operation status of a position p it plays by means of an *activate*(i, p) and *deactivate*(i, p) action, respectively. A position can only be activated if it is inactive and can only be deactivated if it is active. The following two transitions define the organization's response to an *activate* and a *deactivate* action performed by the player of a position.

Transition Rule *Let* $\rho = \langle p, \Sigma, \Gamma, \Pi, \perp \rangle$ *be an inactive position played by agent i. The rule for activating a position is defined as:*

$$\frac{\rho \in \varrho}{\langle \varrho, \sigma_b, \sigma_i, [\mathit{activate}(i, p)] : \epsilon_{in}, \epsilon_{out} \rangle \longrightarrow_{org} \langle \varrho', \sigma_b, \sigma_i, \epsilon_{in}, \epsilon_{out} \rangle} \qquad (\mathit{pactivate})$$

where $\varrho' = (\varrho \setminus \{\rho\}) \cup \{\langle p, \Sigma, \Gamma, \Pi, \top\rangle\}$

Transition Rule *Let $\rho = \langle p, \Sigma, \Gamma, \Pi, \top\rangle$ be an active position played by agent i. The rule for deactivating a position is defined as:*

$$\frac{\rho \in \varrho}{\langle \varrho, \sigma_b, \sigma_i, [deactivate(p)] : \epsilon_{in}, \epsilon_{out}\rangle \longrightarrow_{org} \langle \varrho', \sigma_b, \sigma_i, \epsilon_{in}, \epsilon_{out}\rangle} \qquad (pdeactivate)$$

where $\varrho' = (\varrho \setminus \{\rho\}) \cup \{\langle p, \Sigma, \Gamma, \Pi, \bot\rangle\}$

An agent can inspect and alter the mental state of a position it has enacted. In particular, an agent can inspect and update its beliefs and goals. The next transition rules define the inspection and modification of positions. The performance of action $test(i, p, \phi)$ indicates that the player i of position p performs test ϕ on position p. The organizational artifact responds to such an action by sending the agent a message with the resulting substitution θ of the test, if ϕ is derivable and $\bot$ otherwise. Special substitution ϵ is used to indicate that ϕ is not derivable from the position's mental state. Note that a player can only perform a test on a position's mental state and not on the brute and institutional facts.

Transition Rule *Let $\rho = \langle p, \Sigma, \Gamma, \Pi, f\rangle$ be a position played by agent i and let ϕ be a* ⟨test⟩ *expression containing solely* ⟨belquery⟩ *and* ⟨goalquery⟩ *expressions. The rule for performing an internal test on a position is defined as:*

$$\frac{\rho \in \varrho}{\langle \varrho, \sigma_b, \sigma_i, [test(i, p, \phi)] : \epsilon_{in}, \epsilon_{out}\rangle \longrightarrow_{org} \langle \varrho, \sigma_b, \sigma_i, \epsilon_{in}, \epsilon_{out} : [\phi']\rangle} \qquad (ptest)$$

where $\phi' = \begin{cases} Send(p, i, inform, test(\phi, \theta)) & \text{if } (\Sigma, \Gamma, \emptyset, \emptyset) \models_t \phi\theta \\ Send(p, i, inform, test(\phi, \bot)) & \text{otherwise} \end{cases}$

A goal ϕ can be delegated to a position p by its player by means of an action $adoptg(i, p, \phi)$. Dropping a goal proceeds similarly by action $droptg(i, p, \phi)$. The transition rule defined below only works for the addition and removal of atomic formulae. This most basic operation we believe should be supported at the very least. Other operations for adding and removing goals, as for example can be found in (Dastani, 2008), may be introduced depending on the particular structure of the goal base. Goals can always be delegated to and dropped from a position by its enacting agent. This is specified by the following rule.

Transition Rule *Let $\rho = \langle p, \Sigma, \Gamma, \Pi, f\rangle$ be a position played by agent i and let $\alpha = adoptg/dropg(i, p, \phi)$ with ϕ a ground* ⟨goal⟩ *expression. The rule for adopting/dropping a goal to/from a position is defined as:*

$$\frac{\rho \in \varrho}{\langle \varrho, \sigma_b, \sigma_i, [\alpha] : \epsilon_{in}, \epsilon_{out}\rangle \longrightarrow_{org} \langle (\varrho \setminus \{\rho\}) \cup \{\rho'\}, \sigma_b, \sigma_i, \epsilon_{in}, \epsilon_{out}\rangle} \qquad (pmodg)$$

where $\rho' = \begin{cases} \langle p, \Sigma, \Gamma \cup \{\phi\}, \Pi, f\rangle & \text{if } \alpha = adoptg(p, \phi) \\ \langle p, \Sigma, \Gamma \setminus \{\phi\}, \Pi, f\rangle & \text{if } \alpha = dropg(p, \phi) \end{cases}$

An agent i can also alter the belief base of a position p it plays by adding or removing a belief atom ϕ with actions $adoptb(i,p,\phi)$ and $dropb(i,p,\phi)$, respectively. The following transition rule defines the addition and deletion of beliefs to a position. For updating the belief base we use consistency preserving update function $\oplus$. The actual implementation of this operator depends on the representation of the belief base that comes with the specific agent-oriented programming language. Note that similar to the goal update actions, here we define the the most basic action (i.e. only supporting removal and addition of belief atoms) that should be supported for updating a position's belief base.

Transition Rule *Let $\rho = \langle p, \Sigma, \Gamma, \Pi, f\rangle$ be a position played by agent i and let $\alpha = adoptb/dropb(i,p,\phi)$ in which ϕ is a ground atom. The rule for adopting/dropping a belief to/from a position is defined as:*

$$\frac{\rho \in \varrho}{\langle \varrho, \sigma_b, \sigma_i, [\alpha] : \epsilon_{in}, \epsilon_{out}\rangle \longrightarrow_{org} \langle (\varrho \setminus \{\rho\}) \cup \{\rho'\}, \sigma_b, \sigma_i, \epsilon_{in}, \epsilon_{out}\rangle} \qquad (pmodb)$$

$$\text{where } \rho' = \begin{cases} \langle p, \Sigma \oplus \{\phi\}, \Gamma, \Pi, f\rangle & \text{if } \alpha = adoptb(i,p,\phi) \\ \langle p, \Sigma \setminus \{\phi\}, \Gamma, \Pi, f\rangle & \text{if } \alpha = dropb(i,p,\phi) \end{cases}$$

When the agent is finished with interacting with a position it plays, it may de-enact that position by performing an action $deact(i,p)$. Similar to enactment, a position can only be de-enacted under a priori specified conditions. The conditions under which a player may leave its positions are dictated by the position's destructors. A destrcutor also dictates the actions that should be performed to leave the system in a valid state. If the de-enactment of the position is allowed, it is de-instantiated by executing its destructor and the position is removed from the system. As we have seen before with the constructor, also the whole destructor is executed in one computation step. Upon successful de-enactment of the position the institutional fact $rea(i,r,p)$ denoting that agent i plays role r via position p is retracted from the institutional facts. A position can only be de-enacted when it its inactive. This rule is meant to replace our previsouly defined rule for de-enactment *deact* as defined in chapter 3.

Transition Rule *Let ρ be a position played by agent i and let* `destruct() <-` Φ $\mid$ π *be a destructor of ρ. Then the rule for role de-enactment is defined as:*

$$\frac{\begin{array}{c}\rho = \langle p, \Sigma, \Gamma, \Pi, \bot\rangle \quad \rho \in \varrho \quad (\Sigma, \Gamma, \sigma_b, \sigma_i) \models_t \Phi\theta \\ \langle \{\rho'\}, \sigma_b, \sigma_i, \epsilon_{in}, \epsilon_{out}\rangle \longrightarrow^*_{org} \langle \{\rho''\}, \sigma'_b, \sigma_i, \epsilon_{in}, \epsilon_{out}\rangle\end{array}}{\begin{array}{c}\langle \varrho, \sigma_b, \sigma_i, [deact(i,p)] : \epsilon_{in}, \epsilon_{out}\rangle \\ \longrightarrow_{org} \\ \langle \varrho \setminus \{\rho\}, \sigma'_b, \sigma'_i, \epsilon_{in}, \epsilon_{out} : [Send(p, i, inform, deacted(p))]\rangle\end{array}} \qquad (deact)$$

$$\begin{aligned} \text{where } \rho' &= \langle p, \Sigma, \Gamma, \{\pi\theta\}, \top\rangle \\ \rho'' &= \langle p, \Sigma', \Gamma', \{\}, \top\rangle \\ \sigma'_i &= \sigma_i \setminus \{rea(i,r,p)\} \end{aligned}$$

5.4 Expressing Structural Constraints with Norms

Roles are the cornerstones in defining the structure of an organizational artifact. Part of the definition of the organizational structure is achieved by the ability of a position to communicate with other positions within the organizational artifact. However, the structure of an organization encompasses more than communication alone. We also need means to specify structural constraints, for example, how many times a role may be enacted and which roles are incompatible with one another. In the introduction of this chapter we argued that in existing frameworks such structural constraints can be characterized by two limitations, namely they are: 1) specified at design time and will unconditionally hold at run-time, and 2) are seen as hard constraints that cannot be violated. In this section we further elaborate upon means to define the structure of an organizational artifact by carrying out a preliminary exploration of expressing structural constraints as norms. In particular, we demonstrate that by expressing the structural constraints by the norms presented in chapter 4 we can overcome these limitations.

The norm schemes we introduced in chapter 4 take on the form of conditional obligations and prohibitions. Conditional obligations are expressed as labeled tuples of the form $\phi_l : \langle \varphi_c, O(\varphi_x), \varphi_d \rangle$ with the intuitive reading that "if condition φ_c holds then there is an obligation to establish φ_x before deadline φ_d". A conditional prohibition is expressed as a labeled tuple $\phi_l : \langle \varphi_c, F(\varphi_x), \varphi_d \rangle$ that can be intuitively read as "if condition φ_c holds then it is forbidden to establish φ_x before deadline φ_d." The scheme's condition φ_c and (un)desired state φ_x may refer to institutional facts that model information about role enactment. By modeling conditional obligations and prohibitions that relate to role enactment we can model structural constraints that are dynamically created as obligations and prohibitions are instantiated. Call to mind that in chapter 4 we defined coordination strategies that can be used for norms that can be either regimented (no violations are possible) or enforced (violations are possible and may be sanctioned). This feature allows us to define the organizational structure both as soft-constraints and hard-constraints.

In this chapter we modeled the fact that an agent identified by i plays role r through a position identified by p by an institutional fact $rea(i, r, p)$. Because the norm instances of chapter 4 are supposed to be ground, we cannot write $rea(i, r, p)$ in which p is a variable to denote the fact that agent i plays role r. Therefore, we explicitly define how the binary rea facts can be derived from the ternary rea facts. Note that this type of fact was also used in previous chapters when we did not have positions yet.

Definition 5.5 *Let i and r be constants. Then given a set of institutional facts X we extend the entailment operator $\models$ in the following way:*

- $X \models rea(i, r)$ *iff there exists a ground subst.* θ' *such that* $X \models rea(i, r, p)\theta'$

When using norms to express structural constraints we also need information about how many times this agent plays role r and how often role r is played. To obtain this information from the institutional facts we assume special functions $times(i, r)$ and $times(r)$ to derive this information. Intuitively, the function $times(i, r)$ evaluates to the number of times agent i plays role r. This information can be deduced from the institutional facts by counting all the occurrences of $rea(i, r, p)$ for agent i. Function $times(r, n)$ evaluates to the number of times role r is played, possibly by different agents. This information can be entailed by counting all the occurrences of $rea(i, r, p)$ facts for any agent i.

Having this extra information at our disposal, we are now able to use the norm schemes for defining structural constraints. To start with, we show how to express incompatibility constraints. To define that an agent cannot play role r_1 and role r_2 simultaneously is to write two norm schemes of the form:

$$\phi_{l1} : \langle \varphi_c \wedge rea(i, r_1), F(rea(i, r_2)), \varphi_d \rangle \tag{5.1}$$

$$\phi_{l2} : \langle \varphi_c \wedge rea(i, r_2), F(rea(i, r_1)), \varphi_d \rangle \tag{5.2}$$

meaning that if role r_1 (or r_2) has been enacted under circumstances φ_c it is forbidden to enact role r_1 (or r_2) until φ_d. The norm scheme's condition φ_c and deadline φ_d can thus be used to define the circumstances under which these two roles are incompatible. If this constraint has to hold invariably, that is if these two roles can never be played simultaneously, we simply let $\varphi_c = \top$ and $\varphi_d = \bot$ such that this norm scheme will be immediately instantiated and will stay in effect, because its deadline will never be reached. One might wonder why we did not write this constraint as one norm scheme:

$$\phi_{l1} : \langle \varphi_c, F(rea(i, r_1) \wedge rea(i, r_2)), \varphi_d \rangle \tag{5.3}$$

The reason is that in the current semantics of the norm schemes instantiated obligations and prohibitions should be ground, i.e. contain no variables. The incompatibility constraint can thus only be expressed by the above norm scheme if terms r_1, r_2 and i are constants or, otherwise, also occur in the norm scheme's condition φ_c.

With the special *times* functions we can also express cardinality constraints by means of norm schemes. Provided that terms m, n and r are either constants or also occur in the norm scheme's condition φ_c, we can specify that role r should be played between m and n times by means of the following norm scheme:

$$\phi_l : \langle \varphi_c, F(m > times(r) > n), \varphi_d \rangle \tag{5.4}$$

A cardinality constraint stating that a role r should be played between m and n times by a particular agent i can be expressed as follows:

$$\phi_l : \langle \varphi_c, F(m > times(i, r) > n), \varphi_d \rangle \tag{5.5}$$

Also constraints of a more temporal nature can be expressed by means of norm schemes. Consider for example the following norm scheme:

$$\phi_l : \langle \varphi_c, O(rea(i,r)), \varphi_d \rangle \tag{5.6}$$

that states that under condition φ_c an agent is obliged to enact role r before deadline φ_d. Because we may also use rea facts in the norm scheme's condition, we can also express by a norm scheme of the form:

$$\phi_l : \langle \varphi_c \wedge rea(i,r_1), O(rea(i,r_2)), \varphi_d \rangle \tag{5.7}$$

that an agent who plays role r_1 should under circumstances φ_c eventually (or actually before deadline φ_d) also play role r_2.

These examples show that norm schemes can indeed be used to express an organization's structural constraints. The temporal nature of our norms obtained by the norm scheme's condition and deadline allows for a flexible definition of structural constraints that are invoked at run-time, depending on the circumstances the organizational artifact is in. It is, however, fair to say that not every constraint can be expressed by means of norm schemes. Suppose, for instance, that we would like to define a constraint that some role r_2 should be enacted after an agent has finished playing role r_1. To express this constraint, we need information about when this agent has stopped playing role r_1. At the moment, this information is not available. Storing this information and exploring to what extent other structural constraints can be modelled by means of norm schemes is left for future research.

5.5 Discussion

In this chapter we started with identifying four key properties we believe should be exhibited by a notion of role for efficiently constructing organization-oriented multi-agent systems. Based on the observation that none of the existing role-based approaches we studied in chapter 2 exhibits all four of these properties, we developed a solution that does exhibit all four. These properties thus distinguish our solution from related work on roles. To summarize, our roles are:

- *organization-centric*, meaning that they are an instrinsic part of the organizational artifact instead of being an adjunct of their player.
- *prescriptive*, implying that roles prescribe an expected behavior to their players and in this way guide the players in how to interact with the organization in a meaningful way.
- *employable*, meaning that a player can employ its role by delegating it a task without the need to know how this task is being executed.

- *BDI-oriented*, which means that roles are developed by the same mental attitudes that are often found in agent-oriented programming languages such as beliefs, goals and plans.

Although we deem these properties essential for an implementation of roles in multi-agent systems, also other properties have been identified to classify roles. These properties were identified by Steimann (2000) with the purpose of classifying different approaches to roles in software engineering (see also chapter 2). To give better insights in our approach, we repeat the characteristics that can also be applied on roles from a viewpoint of multi-agent systems (by replacing object by agent) and briefly explain to what extend our proposal exhibits them:

1. *An agent may play different roles simultaneously.* This property is clearly present in our approach; an agent may occupy multiple positions at the same time. In fact, even multiple positions can be activated simultaneously.

2. *An agent may play the same role several times, simultaneously.* In our approach an agent can enact the same role via different positions.

3. *An agent may acquire and abandon roles dynamically.* By performing enact and deact actions, an agent may take on and leave roles at run-time.

4. *The sequence in which roles may be acquired and relinquished can be subject to restrictions.* In our solution, restrictions about the order in which roles may be enacted and de-enacted can be expressed to some extend by the role's constructor and destructor. Also norm schemes may restrict the enactment sequence, as discussed in section 5.4.

5. *Agents of unrelated types can play the same role.* In our approach, we make limited assumptions about the nature of the agents that will play the roles. A role can thus be played by any agent that is capable of performing all the required actions for interacting with a position.

6. *Roles cannot play roles.* In our approach a position is not capable of enacting roles within the artifact or another artifact. Only agents are allowed to enact roles.

7. *A role cannot be transferred from one to another.* In our approach a position cannot be transferred from one player to another player; a player can only de-enact a position.

8. *Roles restrict access.* Players can only access the internals of the organizational artifact and can only communicate with other positions in the system indirectly via their positions. This way, access to the internal state and other positions is restricted by the roles an agent plays.

9. *An agent and its roles share identity.* In our view, a role is organization-centric. This implies that whatever roles an agent enacts, it keeps its own identity. Also the positions that are occupied by some agent have their own identity. An agent and its roles thus do not share identity.

Points 6 and 7 are not yet exhibited by our solution, but do suggest good directions for future research. Indeed, letting roles play roles has also been suggested by Colman *et al.* (2007) as a means for composing different organizations. In our framework this means that a position would be able to play another role in a different (or the same) organizational artifact. In our conference management system, for example, an author position of the reviewing artifact could enact the role of participant in the registration artifact, and register for the conference via this position. By letting positions occupy and interact with other positions we can dynamically compose organizational artifacts. At this point, one might think that this can be easily achieved by letting positions perform enact and deact actions as introduced in this chapter. However, letting positions play positions gives rise to new issues. What, for example, should happen to all the positions that are enacted by a position p in case the agent that plays position p leaves this position? And, what should happen to all positions a position p has enacted when p is (de-)activated? It is interesting to note that answers to these questions can be found in (Dastani and Steunebrink, 2009). In this work Dastani and Steunebrink propose a solution in which agents can be composed of (BDI-oriented) modules, and modules may enact modules. As the authors argue, a module may be considered as an implementation of a role. Indeed, many operations on roles we presented here are inspired by this work. However, an important issue that need to be overcome is that the modules of (Dastani and Steunebrink, 2009) are agent-centered, whereas we argued for an organization-centric view on roles.

Transferring positions from one player to another while preserving state of the position could be useful in case a player is not able to finish its interaction (for example due to failure) before the position can be legally de-enacted. Suppose, for example, that a player of the reviewer role is not able to complete all its reviews, but has already uploaded some of them. In this case it might be beneficial to let another agent take over its task by transferring its position to that agent. This way, all the information, responsibilities and rights of the old player will be transferred to the new player. When transferring positions, mechanisms and policies should be defined for when a position can be transferred from one player to another. We envisage similar constructs for this as the constructor and destructor plans, defining the conditions under which a role may be transferred to another player and the actions that should be taken.

In section 5.4 we carried out a preliminary analysis to what extend structural constraints can be expressed by means of the norm schemes we introduced in chapter 4. This preliminary analysis already showed that norm schemes can indeed express some structural constraints. A further analysis in this direction should reveal to what extend more structural constraints can be expressed and might lead to a reconsideration of the expressiveness of the norm schemes. Further, the organizational structure encompasses more than structural constraints only. Other concepts related to the structure of an organization that are, for example, supported by the Moise framework of Hübner *et al.* (2007) are inheritance of roles, and authority relationships between roles (see (Grossi et al., 2005) for a formal analysis of more structural relationships). Extending our proposal with these structural relationships is left for future research.

Another issue we did not explore in this chapter is how the agents obtain knowledge about which roles can be enacted in an organizational artifact and which goals can be delegated to them. This issue is left for future research. We think the solution of Yellow Pages as used in (Baldoni et al., 2008) is a promising first step in this direction. To end with, we stress the importance of a proof of concept in the form of an implementation. Some notes on a preliminary implementation can be found in appendix A.

5.6 Conclusion

We presented programming constructs for implementing roles and explained them by our running example involving the conference management system. Our notion of roles exhibit four key properties that none of the existing frameworks studied exhibit. We underpinned our syntax by an operational semantics (Plotkin, 1981), allowing us to explain the meaning of the programming constructs in a mathematically rigorous manner, without needing to commit to a specific implementation platform. This way we bring organization-oriented programming closer to agent-oriented programming languages which are often investigated by an operational semantics. We explored the possibility to express structural constraints for expressing the structure of an organization by means of the normative concepts presented in chapter 4. We demonstrated that by means of norm schemes structural constraints can be expressed in a flexible manner.

Chapter 6

Programming Norm Change

In chapter 4 we enhanced our organizational artifacts with a normative dimension for regulating and coordinating the behavior of the interacting agents. Similar to many organizational frameworks (see chapter 2 for an overview), a key characteristic of our norms is that they are designed offline by the system's developer and cannot be changed at runtime. However, there is usually not "one size fits all" specification of the norms. The agents' interactions may have unpredictable outcomes, making it hard for the system's designer to foresee which set of norms fits best for regulating the agents' behavior. Consequently, facing the unpredictable and dynamic nature of the environments organizational artifacts are deployed in, a static view in which the norms are specified at design time and cannot be modified at runtime does often not suffice (Castelfranchi, 2000; Boella et al., 2006; Dignum, 2009b). Modifying norms at runtime increases the system's flexibility to react to the unpredictable interactions that emerge. Suppose, for example, that many reviewers indicate that they will not be able to comply with their obligation to upload their reviews in time. The system (or chair) might react to this by relaxing this norm. A considerable amount of work has been devoted to the theoretical aspects of norm change. Examples are (Alchourron et al., 1985; Boella and van der Torre, 2004; Governatori and Rotolo, 2008)). Thus far, rather less attention has been paid to the practical aspects related to computational mechanisms of norm change.

Similar to the work of Artikis *et al.* (2009), Bou *et al.* (2006), Campos *et al.* (2009), and Oren *et al.* (2010) we focus on a computational mechanism for runtime norm change. Our main contributions are:

- We highlight some of the key challenges involved when changing the set of norms of an organizational artifact at runtime in section 6.1. Each of the challenges will be addressed in this chapter. The identification of the key issues involved allows us to explain how our work advances the state of the art by comparing our work with related work on computational norm change in section 6.6. The discussion is based on the representation of the norms as explained in chapter 4.

- We present the syntax and intuitive semantics of generic programming constructs (section 6.3) allowing the programmer to specify under which circumstances and how the norms may change at runtime.
- We formalize the programming constructs with an operational semantics (Plotkin, 1981) (section 6.4), allowing us to study the proposed constructs in a mathematically rigorous way and show some of the key properties our framework exhibits. Another advantage is that an operational semantics is already close to an interpreter, without committing to a particular implementation language. To our best knowledge, we are the first to study the operational semantics of norm change.
- We propose a construct allowing a programmer to specify which norms are in conflict, e.g. obliging and at the same time forbidding someone to review a paper could be regarded as a conflict. Based on this construct we investigate a mechanism for avoiding normative conflicts when changing the norms at runtime in section 6.5. This has not been done before in respect of a computational mechanism of norm change.

We conclude this paper and give prospects for further research in section 6.7.

6.1 Challenges and Issues for Computational Norm Change

Before we explain the solution we develop for changing the norms at runtime, we first identify what we consider key challenges and issues for runtime norm change. The challenges and issues listed here further motivate the solution we develop and explain how we advance the state of the art with respect to runtime norm change in a computational framework.

Changing Norm Schemes versus Changing Norm Instances

As we have seen in 2, norms are typically specified as conditional sentences defining under which circumstances deontic concepts such as obligations, permissions and prohibitions are established. If, for instance, a reviewer is assigned a paper, an obligation to have uploaded its review for that paper is created. Norms can thus be considered the program specification distinguishing them from the deontic concepts they instantiate. The norms we presented in chapter 4 are no exception to this rule. The norm schemes we presented in 4 take on the form of conditional obligations and prohibitions, that instantiate detached obligations and prohibitions (norm instances) when their condition can be satisfied. Norm change can thus be considered at two levels, namely changes at the level of:

1. the normative specification, i.e. the level norm schemes;
2. the active obligations and prohibitions, i.e. the level of norm instances.

An example of a change at the first level would be relaxing the requirement that each paper should be reviewed by at least three reviewers in case it turns out that there will be too many papers to review. This change affects all norm instances it will instantiated in the future, and thus amounts to a more fundamental change. An example of a change at the second level would be granting the request of a (group of) author(s) who have asked for an extension of the page limit, by altering already instantiated prohibitions. This change only affects some norm instances, without changing the norm scheme. To successfully employ normative frameworks in dynamic, unpredictable environments we argue that changes at both levels should be supported.

When considering change at the level of norm schemes, a key question becomes what happens to the detached obligations and prohibitions that where already created when their underlying norm scheme changes. In some cases it is desirable that the instantiated norm instances remain unaffected, whereas in other cases the associated norm instances should be changed accordingly. Consider, for example, a norm scheme that specifies that conference registrants are obliged to have paid a fee a week before the conference starts. Suppose that for some reason (the costs were higher than expected) we decide to increase this fee, then this increased fee will sensibly only apply for new registrants. In other words, we want the payment obligations that were in effect before the change to remain unaffected. If, however, the payment deadline is extended we might decide to apply this change retroactively. That is to say, the deadline of already existing payment obligations will also be extended.

This issue has been theoretically investigated in the context of modifying legal systems (Governatori and Rotolo, 2008). However, existing work (Artikis, 2009; Bou et al., 2006; Campos et al., 2009; Oren et al., 2010) on computational norm change considers changing the normative specification only. That is to say, the intricacies of how the associated deontic concepts may evolve when a change of their underlying norms occurs is not investigated.

System-Dependency and Enforcement-Independency

Another issue related to computational norm change pertains to who is responsible for deciding under which circumstances and how the norms are changed. Some (e.g. (Castelfranchi, 2000; Artikis, 2009)) consider it the task of the agents to alter the norms. Yet others (e.g. (Bou et al., 2006; Campos et al., 2009)) consider it the responsibility of the organizational framework to autonomously modify the norms. To support a wide variety of application domains, a norm change mechanism ideally empowers both agents and normative framework to initiate norm change. In the solution we develop changes to the norms can be made by external agents as well as the organizational artifact itself.

When external agents that are not a priori known to the system's designer are empowered to change the norms, there should be means to define policies specifying when and how they may change the norms. Firstly, if the system does not put restrictions on who and to which extent norms can be changed the power of the system to regulate the agents' behavior would be compromised. Secondly, deciding when and how norms are changed requires knowledge about and capabilities to reason with norms, fulfillments and violations. The normative framework might benefit from keeping this information private. Most importantly, norm-reasoning capabilities are beyond the competences of typical agency and requiring such a specialized capability limits the types of agents that can interact with the framework, thereby restricting its openness. Therefore, we argue that the normative framework should provide suitable actions by which the agents can change the norms without needing detailed knowledge about their structure. This promotes the principle of encapsulation (a.k.a. data hiding).

The fact that a normative artifact encompasses the policies for when, how and by who the norms may change, implies that present programming languages by which artifacts can be programmed should be enriched with constructs for specifying such policies. We argue that these programming constructs should be such that norm change policies can be defined separately from the norm schemes. That is, the code by which norm change is programmed must be explicitly defined as a separate component rather than entangling it with the code defining the norm schemes. This has the following benefits:

- Separating norm change policies from the norm schemes pertains to the well-known software engineering principle of separation of concerns, promoting readability and manageability of the program code.

- Another advantage is that (if desired) different computation mechanisms could be used for the norm change mechanism without affecting the norm enforcement mechanism and vice versa. In principle, the program that defines how the norms may change could even be plugged in at runtime to deal with unforeseen situations even more flexibly. This is, however, left for future research.

- A final advantage of separating the norm change mechanism from the enforcement mechanism is that it could provide a basis for studying mechanisms allowing agents to reason about norm change. Just like previous point, this is left for future research.

In summary, our approach to norm change is motivated by two key principles of which the first accords with the definition of a normative multi-agent system presented in (Boella et al., 2006) (see also chapter 2) which includes that *"the normative systems specify how and in which extent the agents can modify the norms."*. That is, our norm change mechanism is:

system-dependent: how, who and under which circumstances norms and their instances may be changed must be specified by the normative framework (organizational artifact in our case);

enforcement-independent: the norm change mechanism must be defined separately from the enforcement mechanism.

Conflicting Norm Instances

Suppose that the conference management systems has implemented a rule that a reviewer is forbidden to review papers of (former) colleague's. Furthermore, suppose that `jane` and `john` are colleagues. Now imagine that some reviewer informs the chair that he will not be able to fulfill his obligation to review `john`'s paper, and the chair (possibly not aware that `john` and `jane` are collegueas) decides to change the norm instances by re-assigning this obligation to `jane`. Above situation could be regarded as a conflict between the norm instances; whatever `jane` decides to do – review the paper or not – some norm will be violated. Some have argued that such conflicts between norm instances are ubiquitous, whereas others have argued that normative conflicts do not exist (e.g. standard deontic logic in which we have the D-axiom $\neg(Op \wedge O\neg p)$) (Hansen et al., 2007). When the latter view is adopted, a mechanism should be devised to avoid normative conflicts when norm instances are dynamically added.

It is our goal to devise programming constructs for changing the norms at runtime. Ideally, a programming language is general purpose. Therefore, we do not commit to a particular view on whether the set of norm instances may or may not contain conflicting norm instances. We also do not commit to a particular view on when exactly a set of norm instances is conflicting; there seems to be no consensus on a definition of normative conflict (cf. (Elhag et al., 2000)). Instead, we propose a generic solution in which information about which norm instances are conflicting is assumed to be explicitly present at design time. This information can be (automatically generated) based on some theory of normative conflict, but it can also be defined by the programmer. Additionally, information about which norm instance(s) should be given up in case of conflict is also assumed to be readily available. Getting back to `jane`'s dilemma of our previous example, we can thus decide not to mark these norm instances as conflicting, meaning that `jane` herself should decide which norm to abide by (or should perhaps inform the chair about her dilemma). We could also decide that, say, a prohibition to review a paper of a (former) co-worker should always precede any obligation to review that paper.

In section 6.5 we study a mechanism for avoiding normative conflicts. We deliberately present this mechanism as an incremental extension of the basic norm change mechanism presented in sections 6.3 and 6.4. As we shall see later on, the message is that different strategies exist for resolving normative conflicts and the solution developed in section 6.5 is primarily meant to promote discussion.

6.2 Organizational Artifacts with Changing Norms

As previously explained in chapter 3, we conceive a multi-agent system as consisting of a collection of heterogeneous agents that interact with organizational artifacts. An organizational artifact implements the non-autonomous functionalities that are better implemented by non-agent concepts, such as the functionalities that should be provided by a system implementing a conference management system. The agents exploit the functionality provided by these organizational artifacts to achieve their objectives. They encapsulate a domain specific state and function, which is modeled by a set of *brute facts*. The agents perform actions that change the brute state to interact with the artifact and exploit its functionality. An overview of the internals of an organizational artifact is shown in figure 6.1. In this chapter we will focus on the norm dynamics pertaining to constructs for changing the norms presented in chapter 4 at runtime.

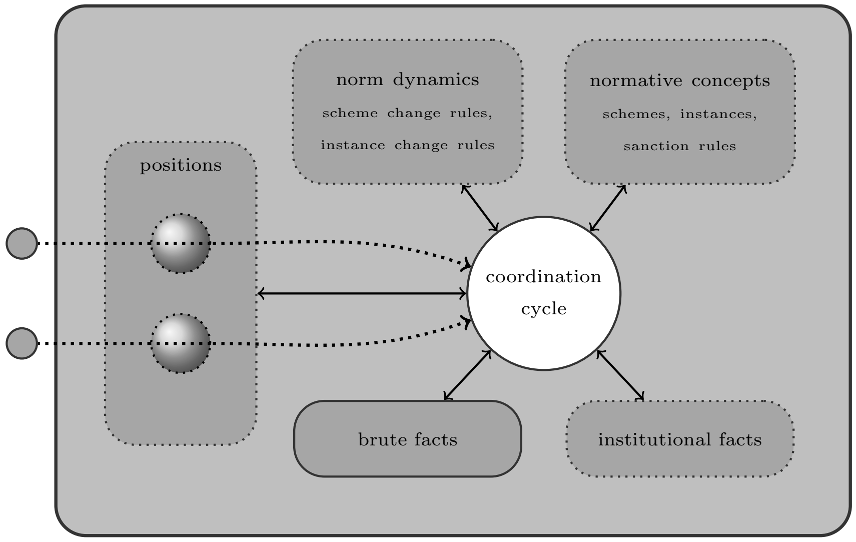


Figure 6.1: A (simplified) conceptual representation of the internals of an organizational artifact. Optional concepts (e.g. norms, positions) are outlined with a dashed border. Solid arrows denote the reading and modification of the concepts as explained in more detail below. Dotted arrows denote actions and messages between agent and artifact. A brief description of all the concepts can be found in chapter 3.

In chapter 4 we extended our notion of organizational artifact with a normative concepts to coordinate the behavior of the possibly unknown interactants and to guide them in interacting with the artifact in a meaningful way. Recall that our normative dimension is programmed by the norm schemes which take on the form of conditional obligations and conditional prohibitions. Conditional obligations are expressed as labeled tuples of the form $\phi_l : \langle \varphi_c, O(\varphi_x), \varphi_d \rangle$ with the intuitive reading that "if condition φ_c holds then there is an obligation to establish φ_x before deadline φ_d". A conditional prohibition is expressed as a labeled tuple $\phi_l : \langle \varphi_c, F(\varphi_x), \varphi_d \rangle$ that can be intuitively read as "if condition φ_c holds then it is forbidden to establish φ_x before deadline φ_d." The norm schemes define under which conditions obligations and prohibitions should be created. For example, if a reviewer is assigned a paper to review, then an obligation to have uploaded a review for that paper is created. The condition of a norm scheme relates to the brute and institutional state of the artifact and whenever its condition is satisfied the artifact instantiates the obligation or prohibition belonging to it, hence the name *norm instance*. Recall that norm instances are of the form $(\phi_l, \mathbb{M}(\varphi_x), \varphi_d)$ in which $\mathbb{M}$ is either an obligation (denoted by O) or an inhibition (denoted by F).

For changing the norms at runtime, we introduce rule-based constructs allowing the artifact's designer to specify how, when and by who the norms may be changed. The rules are specified as an intrinsic part of the organizational artifact. This pertains to the property of system-dependency as explained above. Moreover, the norm-change rules are defined completely separate from the norm schemes which obtain their meaning by the monitoring mechanism. That is to say, the code that defines how, when and by who the norms may be changed is not entangled with the code that defines the norms themselves. This way the norm change mechanism can be guaranteed to be enforcement-independent.

As we mentioned earlier, a change to the normative dimension could be initiated by both agents and the organizational artifact itself. The antecedent of a norm change rule defines the circumstances under which a change should be made to the norms, whereas its consequent lists the changes to be made. The antecedent ranges over both brute and institutional facts, and even includes a test to assess whether particular norm instances are invoked. This way, one the one hand, the rules can be designed to assess the situation of the artifact at runtime and express how the artifact could autonomously change the norms accordingly. On the other hand, because agents can alter the brute facts by the execution of actions, the norm change rules can also be designed to empower agents to change the norms.

We also argued earlier that a norm-change mechanism should facilitate both changes at the level of norm schemes, as well as at the level of norm instances. For this purpose we introduce two types of norm-change rules, namely norm scheme change rules denoted by sc-rules, and norm instance change rules denoted by ic-rules. Because the answer to the question whether the norm instances should evolve accordingly when their underlying norm scheme changes is application specific, we introduce two types of norm scheme change rules. One that does modify the norm instances according to the modification made to their underlying norm scheme, and one type of rule that only changes the norm schemes and leaves their instances unaltered. How this exactly works is explained in more detail below. The relevant steps of the coordination cycle (see chapter 3) that will be introduced in this chapter and concepts involved in changing the norms are summarized by figures 6.2 and 6.3. Recall that in these pictures, arrows from process (depicted as circle) to store (depicted as rounded box) denote the modification of the store's elements, whereas an arrow in the opposite direction denotes the reading of the store's elements.

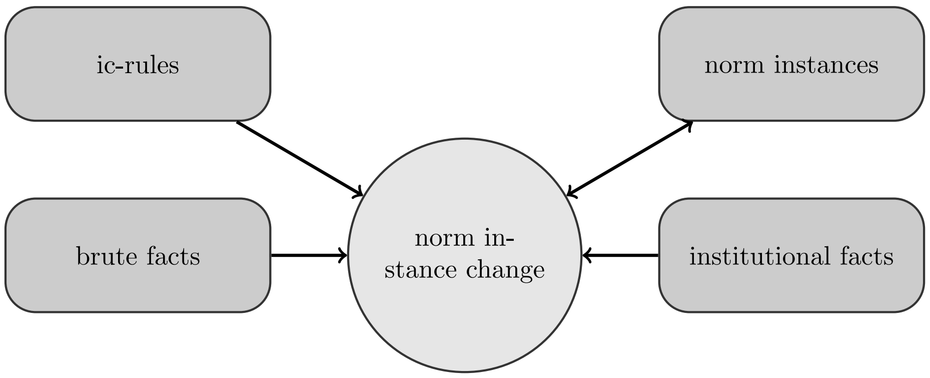


Figure 6.2: Ic-rules specify when, who and how the norm instances may be changed. If the condition of the rule pertaining to brute facts, institutional facts and norm instances that are in effect is satisfied, the norm instances are changed as described by the rule's consequent.

6.3 Programming Norm Change

To facilitate runtime norm change, we introduce rule-based constructs allowing the system's designer to specify how, when and by who the norms may be changed. More concretely, we introduce norm scheme change rules for altering the norm schemes and norm instance change rules for modifying the active obligations and prohibitions. The grammar of ic-rules and sc-rules is shown in figure 6.4. The elementary parts of the syntax are explained in table 6.1. Both types of rules consist of a precondition that describes under which circumstances the rule is applicable and a consequent that specifies the changes to be made. The preconditions of the rules can range over brute facts, institutional facts and norm instances that are in effect. This way we can empower the artifact to change the norms, but also empower agents to modify the norms as agents can change the brute facts via the actions they perform. Both types of rules will be explained in more detail below.

In explaining the norm change rules we use some norms of the conference management system example that were introduced in chapter 4. For the sake of simplicity, we assume that the unique labels of the norm schemes of chapter 4 do

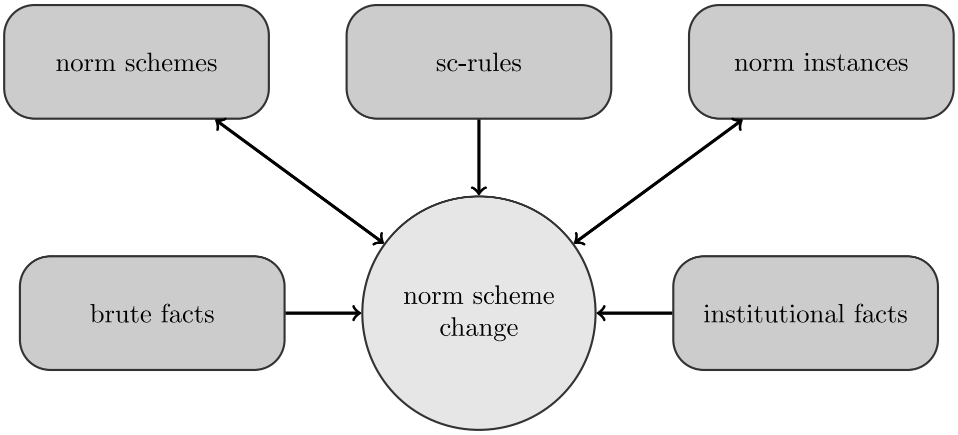


Figure 6.3: Sc-rules specify when, who and how the norm schemes may be changed. If the condition of the rule pertaining to brute facts, institutional facts and norm instances that are in effect is satisfied, the norm schemes are changed as described by the rule's consequent. A change in the norm schemes might also change the instances of the norm schemes that are modified.

not contain variables. By doing so we avoid the need for additational functions to uniquely identify a norm scheme. Because of this assumption, and to promote readability, the norm schemes of the example of chapter 4 that are also used in this chapter are repeated in code fragment 6.1. The first norm states the obligation for reviewers to have uploaded their reviews in time, i.e. before the collect phase starts. The second norm pertains to the page limit, and the third norm states that each paper should be reviewed by at least three reviewers.

6.3.1 Changing Norm Instances

The ic-rules offer a fine-grained mechanism to change one or more norm instances without changing their underlying norm schemes. They specify under which conditions and how changes to the norm instances are made. They are expressed as rules of the form $\beta \Rightarrow [ni_0, ..., ni_n][ni'_0, ..., ni'_m]$ with the intuitive reading that under circumstances β norm instances $ni_0, ..., ni_n$ are to be retracted and norm instances $ni'_0, ..., ni'_m$ are to be asserted. The rule's precondition ranging over brute facts, institutional facts and norm instances thus describes when the norm instances should be modified. How they should be modified is specified by the rule's consequent.

Examples of these ic-rules in the context of our running example are given in code fragment 6.2. Suppose that a reviewer `R1` informs the chair that he will not be able to fulfill his obligation to review a paper, then the chair might

$\langle$b−atom$\rangle$	a first-order atom denoting a brute fact. The special facts starting with predicate symbol *viol*, *obey* and *rea* (their meaning to be explained later on) are excluded from the set of brute facts.
$\langle$r − atom$\rangle$	a first-order atom of the form $rea(i, r)$ (or $rea(i, r, p)$ when positions are used) in which r denotes a role and i the agent playing it (and p the identifier of the position through which the role is enacted); “rea” is short for role enacting agent.
$\langle$i − atom$\rangle$	a first-order atom of the form $viol(\phi_l)$ or $obey(\phi_l)$ with ϕ_l a $\langle$label$\rangle$ identifying a norm. The first denotes a violation of norm labeled ϕ_l whereas the latter denotes the obedience to the norm.

Table 6.1: Elementary syntactical constructs.

```
⟨ic − rule⟩    = ⟨ant⟩ "=>" ⟨ic − cons⟩;
⟨ant⟩          = ⟨b − lit⟩ | ⟨i − lit⟩ | ⟨ni⟩ | ⟨ant⟩ "and" ⟨ant⟩;
⟨ni⟩           = "(" ⟨id⟩ "," ⟨OP⟩ "," ⟨ddln⟩ ")";
⟨ic − cons⟩    = ⟨ic − list⟩ ⟨ic − list⟩;
⟨ic − list⟩    = "[" ⟨ni⟩ { "," ⟨ni⟩ } "]" | "[ ]";
⟨sc − rule⟩    = ⟨ant⟩ "=>" ⟨sc − cons⟩ | ⟨ant⟩ "=>*" ⟨sc − cons∗⟩;
⟨sc − cons⟩    = ⟨sc − list⟩ ⟨sc − list⟩;
⟨sc − cons∗⟩   = ⟨sc − list⟩ ⟨sc − list∗⟩;
⟨sc − list⟩    = "[" ⟨norm⟩ { "," ⟨norm⟩ } "]" | "[ ]";
⟨sc − list∗⟩   = "[" ⟨norm∗⟩ { "," ⟨norm∗⟩ } "]" | "[ ]";
⟨norm∗⟩        = ⟨norm⟩ | "nil";
⟨b − lit⟩      = "true" | ⟨b − atom⟩ | "not" ⟨b − atom⟩;
⟨i − lit⟩      = "true" | ⟨r − atom⟩ | ⟨i − atom⟩ | "not" ⟨r − atom⟩ |
               | "not" ⟨i − atom⟩;
```

Figure 6.4: EBNF grammar of norm change rules.

decide to reassign this paper to another reviewer, say R2. In this case the obligation for reviewer R1 is transferred to reviewer R2, which boils down to removing the obligation of R1 and creating a new one for R2. To give the new reviewer enough time to fulfill its obligation the deadline will be set to the notification phase instead of the collect phase in which reviews are collected. Consider an action `reassign(R1,PId,R2)` by which the chair can reassign paper `PId` from reviewer R1 to R2, modifying the brute state such that the fact `assigned(R1,PId)` is retracted and a fact `reassigned(R1,PId,R2)` is asserted. Then transferring the obligation from reviewer R1 to reviewer R2 is taken care of by the first ic-rule.

The first ic-rule is an example of a norm modification that is initiated by an agent. To illustrate a change of the norms on the decision of the normative framework consider the second ic-rule. Recall the norm of code fragment 6.1 that specifies that at the start of the collect phase at least three reviews should be

Code fragment 6.1 Norm schemes repeated from chapter 4.

```
review-due:
  < phase(review) and assigned(R,PId)
  , O(review(R,PId))
  , phase(collect) >

pl1:
  < phase(submission) and abstract(A,PId)
  , F(pages(PId) > 15)
  , phase(review) >

min-reviews:
  < phase(submission) and paper(PId)
  , O(nrReviews(PId) ≥ 3)
  , phase(collect) >
```

Code fragment 6.2 Example norm instance change rules.

```
reassigned(R1,PId,R2) and (review-due, O(review(R1,PId), phase(P))
⇒
[(review-due,O(review(R1,PId)),phase(collect))]
[(review-due,O(review(R2,PId)),phase(notification))]

phase(review) and 3*nrPapers()/nrReviewers()>5 and
(min-reviews, O(nrReviews(PId)≥3), phase(P))
⇒
[(min-reviews, O(nrReviews(PId)≥3), phase(P))]
[(min-reviews, O(nrReviews(PId)≥2), phase(P))]
```

uploaded for each paper. Assume that this implies that given the actual amount of uploaded papers and reviewers each reviewer would be assigned more than five papers. Suppose that the system reacts to this observation by relaxing the requirement of three reviews per paper by only requiring two. This is expressed by the second ic-rule that will modify all `min-reviews`' instances. Note that the variables of the ic-rule are thus implicitly universally quantified in the widest scope. Because (for the sake of the example) `min-reviews` only instantiates obligations in the submission phase there is no need for modifying the norm scheme also.

6.3.2 Changing Norm Schemes

Whereas ic-rules are used for altering the norm instances, sc-rules are used for modifying the norm schemes. The sc-rules take on a similar form as the ic-rules. More specifically, they are rules of the form $\beta \Rightarrow [ns_i, ..., ns_n][ns'_0, ..., ns'_m]$ in which β is a precondition that specifies when the norm schemes should be changed. How the norm schemes should be changed is specified by the two lists of the rule's consequent. The first list contains the norm schemes that are to be removed, whereas the second list contains the norm schemes that are to be added.

As discussed earlier, in some cases it is desirable that the instantiated norm

Code fragment 6.3 Example norm scheme change rule.

```
nrPapers() > 10 and nrViolations(pagelimit)/nrPapers() > 0.25
⇒*
[pl1:⟨phase(submission) and
       abstract(A,PId),F(pages(PId)>15),phase(review)⟩,
 pl1:⟨phase(submission) and
       abstract(A,PId),F(pages(PId)>15),phase(review)⟩ ]
[pl2:⟨phase(submission) and
       abstract(A,PId),F(15<pages(PId)≤17),phase(review)⟩,
 pl3:⟨phase(submission) and
       abstract(A,PId),F(pages(PId)>17),phase(review)⟩ ]
```

instances remain unaffected when their underlying norm scheme changes, whereas in other cases the associated norm instances should be modified accordingly. For this reason, we propose two types of norm scheme change rules. One that leaves the instantiated norm instances unaltered when their underlying norm scheme is changed and one that revises the associated norm instances accordingly. We name the first type of norm scheme change *instance-preserving* and the latter *instance-revising*. To distinguish between the two rules, the arrow of the instance-revising rules will be annotated with an asterisk, i.e. will take on the form $\Rightarrow_*$.

Updating a norm scheme is to delete the original norm scheme, say ns, and replacing it by a new one, say ns'. If an instance-preserving update is performed, all the instances of ns will remain intact. If, however, an instance-revising update is performed this means that each instance of ns is removed and transformed into an instance of ns'. We thus need to know by which norm scheme a scheme should be replaced. We relate a norm scheme with the norm scheme it should transform into by the position they respectively have in the retract and assert list, visualized:

$$\begin{array}{rl} \beta \Rightarrow_* & [\ \ ns_0,\ ns_1,\ ...,\ ns_n\ \ ] \\ & \quad\ \downarrow \quad \downarrow \qquad\ \ \downarrow \text{ replaced by} \\ & [\ \ ns'_0,\ ns'_1,\ ...,\ ns'_n\ \ ] \end{array}$$

As demonstrated by the following example, the transformation of norm schemes is not necessarily one on one. In fact, a norm scheme might evolve into multiple norm schemes and multiple norm schemes might be merged into one. If one only wants to remove a norm scheme ns together with all its associated instances, special element `nil` can be used to transform ns into an empty norm scheme.

Suppose that it is observed that the norm scheme `pl1` of code fragment 6.1 is often violated, e.g. more than ten papers have been uploaded of which more than 25% violates the page limit norm.[1] In reaction we could, for example, decide to allow authors to pay for an additional two pages and reject papers which exceed the limit by more than two pages. To be able to distinguish between violations that stay within the boundary of two pages and violations by more than two pages, we replace the `pagelimit` norm scheme by two norm schemes `pl2` and

[1]Even though we assume this information to be available, in chapter 4 no history about violations is recorded. Extending the enforcement mechanism to store this information is beyond the scope of this chapter.

pl3 as listed in code fragment 6.3. Because the pl1 norm scheme evolves into two norm schemes it occurs twice in the removal list. To illustrate how the norm instances of norm scheme pl1 are transformed by the application of this sc-rule consider the norm instance:

$$(\mathtt{pl1}, \mathtt{F}(\mathtt{pages}(547) > 15), \mathtt{phase}(\mathtt{review})) \tag{6.1}$$

that was instantiated out of norm scheme pl1 using substitution {PId/547}. To transform above norm instance into norm instances of pl2 and pl3 we use this same substitution. That is to say, we apply the substitution {PId/547} on the norm schemes pl2 and pl3, thereby re-instantiating norm scheme pl1. After application of the sc-rule instance above norm instance will thus be transformed into the two norm instances:

$$(\mathtt{pl2}, \mathtt{F}(15 < \mathtt{pages}(547) \leq 17), \mathtt{phase}(\mathtt{review})) \tag{6.2}$$

$$(\mathtt{pl3}, \mathtt{F}(\mathtt{pages}(547) > 17), \mathtt{phase}(\mathtt{review})) \tag{6.3}$$

6.4 Operational Semantics

In chapter 4 we explained that a norm instance is instantiated from its norm scheme when its condition is derivable from the brute and institutional state for some substitution of its formal parameters. Instantiating a norm scheme is to apply this substitution on it resulting in a norm instance as defined in chapter 4. Because norm instantation plays an important role in the definition of the semantics of the norm change rules we repeat the definition of norm instantiation here. Henceforth, to denote the set of all variables $\overline{v}$ that occur in an unground norm instance or formula ϕ we write $\phi(\overline{v})$.

Definition 6.1 (Norm Instantiation) *Given norm scheme ns of the following form $\phi_l : \langle \varphi_c(\overline{v_1}), \mathbb{M}(\varphi_x(\overline{v_2})), \varphi_d(\overline{v_3})\rangle$ in which $\overline{v_2} \cup \overline{v_3} \subseteq \overline{v_1}$ and a ground substitution θ for the variables the function inst that instantiates a norm instance from ns is defined as*[2]*:*

$$inst(ns, \theta) = (\phi_l, \mathbb{M}(\varphi_x(\overline{v_2})), \varphi_d(\overline{v_3}))\theta$$

Recall that the restriction on the variables is to assure norm instances to be ground, otherwise it would not be clear if the variables occurring in them should be universally or existentially quantified (see also chapter 4). Although we assume ground substitutions in the previous definition and some that will follow, we introduce the notion of well-formedness of a norm change rule. Later we show that given this restriction and the semantics provided below the norm instances indeed remain ground.

[2]Remember that in chapter 4 we annotated prohibitions with a flag to denote if the obligation has been violated in the past. Here we ommit this annotation for the sake of simplicity.

Definition 6.2 (Well-formedness) *We say an ic-rule that takes on the form* $\beta(\overline{x}) \Rightarrow [ni_0(\overline{y}_0), ..., ni_n(\overline{y}_n)][ni'_0(\overline{z}_0), ..., ni'_m(\overline{z}_m)]$ *is well-formed iff all the variables that occur in the rule's consequent also appear in its condition, i.e.* $\bigcup_{0 \leq j \leq n} \overline{y}_j \cup \bigcup_{0 \leq k \leq m} \overline{z}_k \subseteq \overline{x}$.

We say an instance-revising sc-rule of the form $\beta \Rightarrow_* [ns_0, ..., ns_n][ns'_0, ..., ns'_n]$ *is well-formed iff each norm scheme* $ns_j = l : \langle \varphi_c, \mathbb{M}(\varphi_x(\overline{v})), \varphi_d(\overline{w})\rangle$ *and each norm scheme* $ns'_j = l' : \langle \varphi'_c, \mathbb{P}'(\varphi'_x(\overline{x})), \varphi'_d(\overline{y})\rangle$ *for* $0 \leq j \leq n$ *we have* $\overline{x} \cup \overline{y} \subseteq \overline{v} \cup \overline{w}$.

It should be noted that this restriction only guarantees the instances to remain ground in the context of the norm change mechanism. We are now in a position to define the configuration of an organizational artifact extended with norm change rules.

Definition 6.3 (Artifact Configuration) *An organizational artifact configuration is a tuple of the form* $\langle \sigma_b, \sigma_i, \Delta, \delta, R_{ic}, R_{sc}\rangle$ *with:*

- σ_b *a set of ground brute facts;*
- σ_i *a set of ground insitutional facts;*
- Δ *a set of norm schemes;*
- δ *a set of ground norm instances;*
- R_{ic} *a set of well-formed ic-rules;*
- R_{sc} *a set of well-formed (instance-revising and instance-preserving) sc-rules.*

A configuration $\langle \sigma_b, \emptyset, \Delta, \emptyset, R_{ic}, R_{sc}\rangle$ *specified by a program s.t.* σ_b *is characterised by the program's facts component,* Δ *defined by the program's norms component,* R_{ic} *and* R_{sc} *respectively defined by the program's ic-rules and sc-rules component is called an initial artifact configuration.*

Henceforth, we assume an artifact configuration $\langle \sigma_b, \sigma_i, \Delta, \delta, R_{ic}, R_{sc}\rangle$. The components R_{ic} and R_{sc} will be ommitted, because they will not change during computation. In applying the rules of norm change their precondition needs to be evaluated. Recall that the condition ranges over brute facts, institutional facts and norm instances. We define the entailment for preconditions as follows.

Definition 6.4 (Entailment) *Let* ϕ *be a (brute or institutional) literal,* $(l, \mathbb{M}(\varphi_x), \varphi_d)$ *a norm instance, and* $\psi_1(\overline{x})$, $\psi_2(\overline{y})$ *a rule's antecedent. Given substitutions* θ, θ_1 *and* θ_2*, the entailment relation* $\models_t$ *that evaluates rule condition expressions w.r.t. sets of brute facts, institutional facts and norm instances* $(\sigma_b, \sigma_i, \delta)$ *is defined as:*

- $(\sigma_b, \sigma_i, \delta) \models_t (\phi)\theta$ iff $\phi\theta \in (\sigma_b \cup \sigma_i)$
- $(\sigma_b, \sigma_i, \delta) \models_t (l, \mathbb{M}(\varphi_x), \varphi_d)\theta$ iff $(l, \mathbb{M}(\varphi_x), \varphi_d)\theta \in \delta$

- $(\sigma_b, \sigma_i, \delta) \models_t (\psi_1(\overline{x})$ and $\psi_2(\overline{y}))\theta$ iff $\exists\theta_1 : [\theta_1 = \theta|\overline{x}$
  and $(\sigma_b, \sigma_i, \delta) \models_t \psi_1(\overline{x})\theta_1$ and $\exists\theta_2 : [\theta_2 = \theta|(\overline{y} \setminus \overline{x})$
  and $(\sigma_b, \sigma_i, \delta) \models_t \psi_2(\overline{y})\theta_1\theta_2]]$

in which "|" is to be read as "restricted to the domain".

An ic-rule is applicable when its precondition can be entailed for some substitution θ. Applying an ic-rule is then to apply this substitution that satisfies the precondition on each element of the rule's retraction and assertion list. The result is a set of norm instances to be removed and a set of norm instances to be asserted. The artifact's norm instances will then be updated by first removing the first set and then asserting the latter set. The following transition rule defines the application of an ic-rule. We annotate the transition (and the transitions that will follow) with information about the which rule is applied and the substitution for which the antecedent was applicable. This is not essential for the semantics, but we use it later on to ease notation.

Transition Rule *The transition rule for applying an ic-rule is defined as:*

$$\frac{r = (\beta \Rightarrow [ni_0, ..., ni_n][ni'_0, ..., ni'_m]) \in R_{ic} \quad (\sigma_b, \sigma_i, \delta) \models_t \beta\theta\}}{\langle \sigma_b, \sigma_i, \Delta, \delta \rangle \xrightarrow{r,\theta}_{org} \langle \sigma_b, \sigma_i, \Delta, \delta' \rangle} \qquad (ic)$$

w*ith* $\delta' = (\delta \setminus \{ni_0\theta, ..., ni_n\theta\}) \cup \{ni'_0\theta, ..., ni'_m\theta\}$

Recall that the application of an instance-preserving rule does only affect the norm schemes, leaving their associated instances unaltered. The transition rule that defines the application of instance-preserving sc-rules is defined in a similar manner as that of the ic-rules. The difference is that now the norm schemes are updated instead of the norm instances. Unlike ic-rules, we assume that each variable that occurs in an (instance-revising or instance-preserving) sc-rule's antecedent does not occur in its consequent. We take this assumption to not unnecessarily complicate the semantics and the proofs of the propositions of the (in particular instance-revising) sc-rules.

Transition Rule *The transition rule for applying an instance-preserving sc-rule is defined in the following manner:*

$$\frac{r = (\beta \Rightarrow [ns_0, ..., ns_n][ns'_0, ..., ns'_m]) \in R_{sc}(\sigma_b, \sigma_i, \delta) \models_t \beta\theta}{\langle \sigma_b, \sigma_i, \Delta, \delta \rangle \xrightarrow{r}_{org} \langle \sigma_b, \sigma_i, \Delta', \delta \rangle} \qquad (ipsc)$$

w*ith* $\Delta' = (\Delta \setminus \{ns_1, ..., ns_n\}) \cup \{ns'_1, ..., ns'_m\}$

The application of an instance-revising sc-rule not only involves updating the norm schemes, but also involves updating the norm instances that have been instantiated out of them. Recall that we use the substitutions that are used for creating instances of the norm scheme to be modified, say *ns*, for creating new norm instances of the norm scheme ns is replaced with. We need to know which norm instances it has created and which substitution of the variables has been used in creating them.

Definition 6.5 (Instances) *Let ns be a norm scheme and S be a set of norm instances. The function* I *that evaluates to all norm instances in S which are instances of ns together with their associatated substitutions, is defined as:*

$$\mathrm{I}(ns, S) = \{(ni, \theta) \mid ni \in S \text{ and } \mathit{inst}(ns, \theta) = ni \text{ and } \theta \text{ a ground substitution}\}$$

Remember that the consequent of an instance-revising scheme change rule takes on the form $[ns_0, ..., ns_n][ns'_0, ..., ns'_n]$. When an instance-revising sc-rule is applicable, this intuitively means that each norm scheme ns_i for $0 \leq i \leq n$ will be replaced by norm scheme ns'_i and all instances of ns_i will be re-instantiated into instances of ns'_i. Applying this instance-preserving sc-rule then boils down to: 1) removing each norm scheme ns_i from the normative artifact; 2) asserting each norm scheme ns'_i to the normative artifact; 3) removing all instances of each norm scheme ns_i; and 4) instantiating each ns'_i with the same substitutions that were used in instantiating the associated norm instances of ns_i.

Transition Rule *The transition rule for applying an instance-revising sc-rule is defined as follows:*

$$\frac{r = (\beta \Rightarrow_* [ns_0, ..., ns_n][ns'_0, ..., ns'_n]) \in R_{sc} \quad (\sigma_b, \sigma_i, \delta) \models_t \beta\theta}{\langle \sigma_b, \sigma_i, \Delta, \delta \rangle \xrightarrow{r}_{org} \langle \sigma_b, \sigma_i, \Delta', (\delta \setminus \delta^-) \cup \delta^+ \rangle} \qquad (irsc)$$

$$\begin{aligned} \text{with } \Delta' &= (\Delta \setminus \{ns_0, ..., ns_n\}) \cup (\{ns'_0, ..., ns'_n\} \setminus \{\texttt{nil}\}) \\ \delta^- &= \textstyle\bigcup_{0 \leq j \leq n} \{ni \mid (ni, \theta') \in \mathrm{I}(ns_j, \delta)\} \\ \delta^+ &= \textstyle\bigcup_{0 \leq j \leq n} \{\mathit{inst}(ns'_j, \theta') \mid (ni, \theta') \in \mathrm{I}(ns_j, \delta) \text{ and } ns'_j \neq \texttt{nil}\} \end{aligned}$$

We conclude this section by showing some basic, yet essential, properties the semantics exhibits. These properties summarize the meaning of the norm-change rules and demonstrate they indeed behave as we intuitively explained in previous sections. To ease notation we define some auxiliary operators. Given an ic-rule or sc-rule rule $r = \beta \Rightarrow_{(*)} [\phi_0, ..., \phi_n][\phi'_0, ..., \phi'_m]$ and substitution θ we define:

$$\begin{aligned} \mathit{add}(r)\theta &= \{\phi'_0\theta, ..., \phi'_m\theta\} \\ \mathit{del}(r)\theta &= \{\phi_0\theta, ..., \phi_n\theta\} \\ \mathit{tail}(r) &= [\phi_0\theta, ..., \phi_n\theta][\phi'_0\theta, ..., \phi'_m\theta] \end{aligned}$$

Substitution θ will be omitted whenever empty. This notation allows us to define the following auxiliary operators.

Definition 6.6 (Update, addition and retraction) *Let $\langle \sigma_b, \sigma_i, \Delta, \delta \rangle$ be an artifact configuration. The operators $\oplus$, $\ominus$ and $\circledast$ are defined as follows:*

- $\delta \oplus add(r)\theta = \delta'$ iff $r = \beta \Rightarrow [\,][ni_0, ..., ni_n]$
  and $\langle \sigma_b, \sigma_i, \Delta, \delta \rangle \xrightarrow{r,\theta}_{org} \langle \sigma_b, \sigma_i, \Delta, \delta' \rangle$
- $\delta \ominus del(r)\theta = \delta'$ iff $r = \beta \Rightarrow [ni_0, ..., ni_n][\,]$
  and $\langle \sigma_b, \sigma_i, \Delta, \delta \rangle \xrightarrow{r,\theta}_{org} \langle \sigma_b, \sigma_i, \Delta, \delta' \rangle$
- $(\Delta, \delta) \circledast tail(r) = (\Delta', \delta')$ iff $r = \beta \Rightarrow_* [ns_0, ..., ns_n][ns'_0, ..., ns'_n]$
  and $\langle \sigma_b, \sigma_i, \Delta, \delta \rangle \xrightarrow{r}_{org} \langle \sigma_b, \sigma_i, \Delta', \delta' \rangle$

The following propositions highlight the essential meaning of norm instance change rules. The first two show that instances of the retract list are indeed removed from the set of norm instances and that this set is indeed expanded by the instances of the addition list. The third shows that retracting and consecutively asserting a set of instances yields the same set. It tells us that we can recover a change once made. It is interesting to note the similarity (but not equivalence!) with the success of retraction and expansion, and recovery AGM postulates (Alchourron et al., 1985). The AGM postulates formulate the properties that an update operator should satisfy.

Proposition 6.7 *Given set of norm instances S. We have that:*

1. $S \subseteq (\delta \oplus S)$
2. $S \cap (\delta \ominus S) = \emptyset$
3. *if* $S \subseteq \delta$ *then* $\delta = (\delta \ominus S) \oplus S$

Sketch of Proof. *Follows from the definition of rule ic which adds/removes the whole set and only the whole set of norm instances $\{ni_0\theta, ..., ni_n\theta\}$ to/from δ.*

Similar (trivial) results can be shown for the instance-preserving sc-rules. The propositions pertaining to the update of the instance-revising scheme change rules, however, are more interesting. These are shown below. The first proposition pertains to success of addition, whereas the second pertains to success of retraction. The third proposition could be considered success of re-instantiating the norm instances according to the change of their underlying norm scheme. The fourth proposition can be considered the recovery postulate for an instance-revising update of the norm schemes.

Proposition 6.8 *Given instance-revising sc-rule $r1 = \beta \Rightarrow_* [ns_0, ..., ns_n][ns'_0, ..., ns'_n]$ and its reverse $r2 = \beta \Rightarrow_* [ns'_0, ..., ns'_n][ns_0, ..., ns_n]$ s.t. all norm schemes have disjoint labels and are not* `nil`, *and $(\delta', \Delta') = (\delta, \Delta) \circledast tail(r1)$. Then the following propositions hold:*

1. $add(r1) \subseteq \Delta'$
2. $del(r1) \cap \Delta' = \emptyset$

3. if $inst(ns_j, \theta) \in \delta$ with $0 \leq j \leq n$ and subst. θ then
$inst(ns_j, \theta) \notin \delta'$ and $inst(ns'_j, \theta) \in \delta'$

4. if $del(r1) \subseteq \Delta$ and $add(r1) \cap \Delta = \emptyset$ and $\forall_{0 \leq j \leq n} : \mathrm{I}(ns'_j, \delta) = \emptyset$ then
$(\delta, \Delta) = (\delta', \Delta') \circledast tail(r2)$

Proof. *Let $S = \{ns_0, ..., ns_n\}$ and $S' = \{ns'_0, ..., ns'_n\}$.*

(1 and 2) Follows from rule irsc in which $\Delta' = (\Delta \setminus S) \cup S'$ and S disjoint with S' (by assumption of disjoint labels).

(3) Note that $\delta' = (\delta \setminus \delta^-) \cup \delta^+$ (definition of rule irsc). We have $inst(ns'_j, \theta) \in \delta'$, because $inst(ns'_j, \theta) \in \delta^+$ through 1) $inst(ns_j, \theta) \in \delta$ (by assumption) and 2) $ns'_j \neq$ `nil` (by assumption). We have $inst(ns_j, \theta) \notin \delta'$ since 1) $inst(ns_j, \theta) \in \delta^-$ and 2) $inst(ns_j, \theta) \notin \delta^+$ (assumption of disjoint labels).

(4) Let $(\delta'', \Delta'') = (\delta', \Delta') \circledast tail(r2)$. To prove that $(\delta, \Delta) = (\delta'', \Delta'')$. Rule irsc ensures $\Delta = \Delta''$ because 1) $\Delta' = (\Delta \setminus S) \cup S'$ (rule irsc), 2) $\Delta'' = (\Delta' \setminus S') \cup S$ (rule irsc), 3) $S \subseteq \Delta$ and $S' \cap \Delta = \emptyset$ (by assumption) and 4) $S \cap S' = \emptyset$ (assumption of disjoint labels). To prove $\delta = \delta''$ we show that $ni \in \delta \Leftrightarrow ni \in \delta''$ with ni an instance of ns_j or ns'_j for $0 \leq j \leq n$. Note that $\delta'' = (\delta' \setminus \delta'^-) \cup \delta'^+$ and $\delta' = (\delta \setminus \delta^-) \cup \delta^+$ (by definition of rule irsc). Note that $\mathrm{I}(ns'_j, \delta) = \emptyset$ (by assumption), so for ($\Rightarrow$) we only take $inst(ns_j, \theta) \in \delta$ for ground substitution θ. Rule irsc ensures $inst(ns_j, \theta) \in \delta''$ because 1) $inst(ns'_j, \theta) \in \delta'$ (by definition of δ^+ and $ns'_j \neq$ `nil`), 2) $(inst(ns'_j, \theta), \theta) \in \mathrm{I}(ns'_j, \delta')$ (because of well-formedness) and consequently 3) $inst(ns_j, \theta) \in \delta'^+$ (by definition of δ'^+ and $ns_j \neq$ `nil`). For ($\Leftarrow$) we use contraposition. Assume $inst(ns_j, \theta) \notin \delta$ for ground substitution θ. Rule irsc ensures $inst(ns_j, \theta) \notin \delta''$, because 1) $inst(ns'_j, \theta) \notin \delta$ (by assumption), 2) $inst(ns'_j, \theta) \notin \delta'$ (by definition of δ^+ and disjoint labels) and consequently 3) $inst(ns_j, \theta) \notin \delta'^+$ (by definition of δ'^+). Next, assume $inst(ns'_j, \theta) \notin \delta$ for some substitution θ. Rule irsc ensures that $inst(ns'_j, \theta) \notin \delta''$ because 1) δ'^- is such that all instances of ns'_j are removed and 2) $inst(ns'_j, \theta) \notin \delta'^+$ (by assumption of disjoint labels).

Interestingly, recovery only holds under certain conditions of which the least obvious one is the restriction that all labels are disjoint. (In fact, this restriction is too strong, but keeps the proposition comprehensible.) This excludes consecutively applying a rule with consequents $[ns1, ns2][ns3, ns3]$ and $[ns3, ns3][ns1, ns2]$ (and assume that no variables in the consequent are bound by variables in the rule's antecedent). Suppose $ns1 = l1 : \langle c, O(x(X)), d\rangle$, $ns2 = l2 : \langle c, O(x(X)), d\rangle$ and $ns3 = l3 : \langle c, O(x(X)), d\rangle$. It is left to the reader to check that given set of norm instances $\delta = \{(l1, O(x(a)), d)\}$ this execution yields the set of norm instances $\delta'' = \{(l1, O(x(a)), d), (l2, O(x(a)), d)\}$.

The concept of well-formedness was introduced with the goal of guaranteeing the norm instances to be ground. The following proposition shows that given the semantics for the norm change rules they indeed remain ground.

Proposition 6.9 *Let $\gamma_0 = \langle \sigma_b, \sigma_i, \Delta, \delta\rangle$ be an artifact configuration s.t. all norm*

instances in δ are ground. Then for every trace $\gamma_0 \rightarrow_* \gamma_n$ with $\gamma_n = \langle \sigma'_b, \sigma'_i, \Delta', \delta' \rangle$ it holds that that each norm instance $ni \in \delta'$ is ground.
Proof. *Only transition rules ic and irsc add instances. Rule ic is such that the substitutions that satisfy the antecedent are applied on the norm instances in the consequent. These substitutions are ground because of well-formedness. Rule irsc re-instantiates the instances of each scheme in the retract list by applying the substitution used in creating them on the corresponding scheme in the assert list. Because of well-formedness these substitutions are ground.*

6.5 Towards a Mechanism for Avoiding Normative Conflicts

Hitherto, we did not take the possibility of conflicting norm instances into account when defining the semantics of the norm change rules. Suppose, for example, we regard two norm instances, say $ni = (\varphi_c, F\varphi_x, \varphi_d)$ and $ni' = (\varphi'_c, O\varphi_x, \varphi_d)$, conflicting, because no matter what actions the agents undertake, a violation cannot be avoided[3]. Then there is nothing that guarantees the remaining set of norm instances to be free of conflict when we execute a rule that asserts ni to a set of norm instances already containing ni'.

In this section we explore a mechanism for avoiding conflicting norm instances. We emphasize that multiple strategies exist for avoiding normative conflicts when updating the set of norm instances by means of the norm change rules. Here, we study only one of many possible strategies and study the implications for the properties we have previously shown for the 'standard' semantics. As mentioned earlier, the solution presented here is by no means meant to be definite, but rather to provoke discussion and show the intricacies of defining a mechanism for avoiding normative conflicts.

As pointed out earlier, different opinions exists to what is a normative conflict and if they should be part of a theory of deontic concepts at all. Here, we do not commit to a particular notion of normative conflict. We simply assume that information about when norm instances are conflicting is available to the system. More specifically, we introduce a precedence relation $\succ$ over norm instances. We use this relation in deciding which norm instances should be retained and which should be given up in updating the norm instances. The idea of this relation is twofold. Suppose we have that $(ni, ni') \in \succ$ (also written as $ni \succ ni'$), this intuitively means that: 1) ni and ni' are considered to be conflicting and, consequently, should not be issued at the same time and; 2) ni precedes (or overrules) ni', i.e. norm instance ni is deemed more important than ni'. The notion of our precedence relation can be compared with the notion of epistemic entrenchment; it gives us extra information in determining which norm instances should be given up in case conflicts occur when updating the norm instances.

Definition 6.10 (Normative Precedence Relation) *A conflict relation $\succ$ is a set of pairs*

[3]In chapter 4 it has been shown that in this case a violation is inevitable.

of (possibly unground) norm instances such that:

- *if* $(ni, ni') \in \succ$ *and* $(ni', ni'') \in \succ$ *then* $(ni, ni'') \in \succ$*;*
- *if* $(ni, ni') \in \succ$ *then* $(ni', ni) \notin \succ$ *and* $ni \neq ni'$*.*

In what follows we write $ni \succ ni'$ *to denote that* $(ni, ni') \in \succ$*.*

Note that in specifying $\succ$ the norm instances may contain variables. The intuitive meaning of the conflict relation is as follows. When some norm instances ni and ni' are related by $\succ$, i.e. $ni \succ ni'$, these norm instances are considered to be in conflict and ni has precedence over ni' under every possible substitution θ. Note that $\succ$ is irreflexive and transitive, which together imply asymmetry. In fact, $\succ$ defines a partial order over the norm instances. Also note that we do not commit to any particular definition of when norms are conflicting. The precedence relation indicates which norm instances are in conflict. It can be specified by the programmer, but it can also be generated automatically based on some theory of normative conflict.

Definition 6.11 (Subordination and Normative Conflict) *Given sets of norm instances* S *and* S'*, and a normative precedence relation* $\succ$*, we define the function* $Sub^{\succ}(S, S')$ *to determine all norm instances in* S *that are subordinate to some norm instance in* S'*:*

$$\begin{array}{rcl} Sub^{\succ}(S, S') & = & \{ni \in S \mid ni' \in S' \text{ and } y \succ x \text{ and } x\theta = ni \text{ and } y\theta = ni' \\ & & \quad \text{for some substitution } \theta\} \end{array}$$

We say S *is non-conflicting (under* $\succ$*) iff* $Sub^{\succ}(S, S) = \emptyset$*.*

We aim to avoid normative conflicts when updating a set of norm instances by means of the norm change rules. To define such a semantics, we first need to extend the notion of an organizational artifact to include a conflict relation also.

Definition 6.12 (Extended Organizational Artifact) *An extended (normative) artifact configuration is a tuple* $\langle \sigma_b, \sigma_i, \Delta, \delta, R_{ic}, R_{sc}, \succ \rangle$ *in which* $\sigma_b, \sigma_i, \Delta, \delta, R_{ic}$ *and* R_{sc} *are as defined in definition 6.3, and* $\succ$ *is a normative precedence relation.*

Henceforth, an extended artifact configuration $\langle \sigma_b, \sigma_i, \Delta, \delta, R_{ic}, R_{sc}, \succ \rangle$ is assumed. As before, components that will not change at runtime (viz. R_{ic}, R_{sc}, and $\succ$) will be omitted in defining the operational semantics. This operational semantics is an extension of the transition rules defined in the previous section. The transition rules presented here thus replace the ones previously defined. In defining them we introduce the following conflict-avoiding update function that updates a set of norm instances S by a set of norm instances S' under normative precedence relation $\succ$:

$$Up^{\succ}(S, S') = S \setminus Sub^{\succ}(S, S' \setminus Sub^{\succ}(S', S)) \cup (S' \setminus Sub^{\succ}(S', S))$$

This function thus adds all norm instances from S' that are not weaker than some norm in S, i.e. the set $S' \setminus Sub^{\succ}(S', S)$. Moreover, it removes all norm instances from S that are weaker than some norm that is added, i.e. $Sub^{\succ}(S, S' \setminus Sub^{\succ}(S', S))$. Note that in removing weaker norm instances from S, we must limit ourselves to norm instances that are actually added instead of all norm instances in S'. Otherwise, we might end up with removing too many instances. To see this, suppose we did not limit ourselves only to norms that are added and that we update a set $\{ni_1, ni_2\}$ with $\{ni_3\}$ with $ni_3 \succ ni_1$ and $ni_2 \succ ni_3$. Then ni_3 will not be added because $ni_2 \succ ni_3$, but we also unnecessarily remove ni_1.

We use this update function for re-defining the operational semantics for the norm change rules designed to avoid normative conflicts. Recall that ic-rules as well as instance-revising sc-rules may modify the norm instances. Therefore, the semantics of both rules needs to be redefined to guarantee the absence of normative conflicts. When updating the norm instances by means of an ic-rule, avoiding normative conflicts then boils down to first removing all instances from the retract list and using the conflict-avoiding update function $Up^{\succ}$ to assert the norm instances from the assert list. Additionally, we require that the norm instances that will be asserted by an applicable ic-rule are not in conflict with each other. It should be noted, however, that this requirement could be relaxed by, for example, modifying the update function $Up^{\succ}$ such that conflicting norm instances in S' are also filtered out before asserting them.

Transition Rule *The application of a conflict-avoiding ic-rule is defined as:*

$$\frac{r = (\beta \Rightarrow [ni_0, ..., ni_n][ni'_0, ..., ni'_m]) \in R_{ic} \quad (\sigma_b, \sigma_i, \delta) \models_t \beta\theta \qquad Sub^{\succ}(\delta^+, \delta^+) = \emptyset}{\langle \sigma_b, \sigma_i, \Delta, \delta \rangle \xrightarrow{r,\theta}_{org} \langle \sigma_b, \sigma_i, \Delta, Up^{\succ}(\delta \setminus \delta^-, \delta^+) \rangle} \quad (ic*)$$

with $\delta^- = \{ni_0\theta, ..., ni_n\theta\}$ and $\delta^+ = \{ni'_0\theta, ..., ni'_m\theta\}$

An instance-revising scheme change rule changes the norm instances according to the changes made to their underlying norm schemes. To avoid normative conflicts is to assure that the re-instantiated norm instances do not conflict with already existing norm instances. Hereto, we first remove all instances of all norm schemes that are removed and then use the conflict-avoiding update function $Up^{\succ}$ to assert all re-instantiated norm schemes. Again, we demand that the newly asserted norm instances are not mutually conflicting.

Transition Rule *The application of a conflict-avoiding instance-revising sc-rule is defined as:*

$$\frac{r = (\beta \Rightarrow_* [ns_0, ..., ns_n][ns'_0, ..., ns'_n]) \in R_{sc} \quad (\sigma_b, \sigma_i, \delta) \models_t \beta\theta \qquad Sub^{\succ}(\delta^+, \delta^+) = \emptyset}{\langle \sigma_b, \sigma_i, \Delta, \delta \rangle \xrightarrow{r}_{org} \langle \sigma_b, \sigma_i, \Delta', Up^{\succ}(\delta \setminus \delta^-, \delta^+) \rangle} \quad (irsc*)$$

$$\begin{aligned} with\ \Delta' &= (\Delta \setminus \{ns_0, ..., ns_n\}) \cup (\{ns'_0, ..., ns'_n\} \setminus \{\texttt{nil}\}) \\ \delta^- &= \bigcup_{0 \leq j \leq n} \{ni \mid (ni, \theta') \in \mathrm{I}(ns_j, \delta)\} \\ \delta^+ &= \bigcup_{0 \leq j \leq n} \{inst(ns'_j, \theta') \mid (ni, \theta') \in \mathrm{I}(ns_j, \delta) \text{ and } ns'_j \neq \texttt{nil}\} \end{aligned}$$

The aim of the conflict-avoiding semantics is to guarantee that under the execution of the norm change rules the set of norm instances remains non-conflicting. The following proposition shows that this is indeed the case for the conflict-avoiding strategy.

Proposition 6.13 *Let $\langle \sigma_b, \sigma_i, \Delta, \delta, R_{ic}, R_{sc}, \succ \rangle$ be an extended artifact configuration such that δ is non-conflicting. Then:*

1. *if $\langle ..., \Delta, \delta, ... \rangle \xrightarrow{r,\theta}_{org} \langle ..., \Delta, \delta', ... \rangle$ then δ' is non-conflicting.*
2. *if $\langle ..., \Delta, \delta, ... \rangle \xrightarrow{r}_{org} \langle ..., \Delta', \delta', ... \rangle$ then δ' is non-conflicting.*

Proof. *(1) Note that $\delta' = Up^{\succ}(\delta \setminus \delta^-, \delta^+)$ (by definition of rule ic*). We have that δ' is non-conflicting, i.e. $Sub^{\succ}(\delta', \delta') = \emptyset$ because: 1) δ is non-conflicting (by assumption) and consequently 2) $\delta \setminus \delta^-$ is non-conflicting, and 3) δ^+ is non-conflicting (due to the condition of rule ic*) and 4) $Up^{\succ}$ is constructed such that only norm instances are added to $\delta \setminus \delta^-$ that are not weaker than some norm instance in $\delta \setminus \delta^-$ and all norm instances in $\delta \setminus \delta^-$ that are weaker than some norm instance that is added are removed. We thus have $Sub^{\succ}(\delta', \delta') = \emptyset$.*

(2) Based on a similar reasoning as (1).

In previous section we have shown some basic, yet essential, properties the norm change rules exhibit under the 'standard' semantics. Having introduced an alternative semantics for avoiding normative conflict, we show some similar properties under this new semantics. Hereto, we define update operators similar to those of definition 6.6.

Definition 6.14 (Update, addition and retraction) *Let $\langle \sigma_b, \sigma_i, \Delta, \delta \rangle$ be an extended artifact configuration. The conflict-avoiding operators $\oplus^*$, $\ominus^*$ and $\circledast^*$ are defined as follows:*

- $\delta \oplus^* add(r)\theta = \delta'$ iff $r = \beta \Rightarrow [\,][ni_0, ..., ni_n]$
  and $\langle \sigma_b, \sigma_i, \Delta, \delta \rangle \xrightarrow{r,\theta}_{org} \langle \sigma_b, \sigma_i, \Delta, \delta' \rangle$
- $\delta \ominus^* del(r)\theta = \delta'$ iff $r = \beta \Rightarrow [ni_0, ..., ni_n][\,]$
  and $\langle \sigma_b, \sigma_i, \Delta, \delta \rangle \xrightarrow{r,\theta}_{org} \langle \sigma_b, \sigma_i, \Delta, \delta' \rangle$
- $(\Delta, \delta) \circledast^* tail(r) = (\Delta', \delta')$ iff $r = \beta \Rightarrow_* [ns_0, ..., ns_n][ns'_0, ..., ns'_n]$
  and $\langle \sigma_b, \sigma_i, \Delta, \delta \rangle \xrightarrow{r}_{org} \langle \sigma_b, \sigma_i, \Delta', \delta' \rangle$

Proposition 6.15 *Given set of norm instances S, S' and let δ be non-conflicting. Moreover, let $\mathbb{WF}(S, S') = S \setminus Sub^{\succ}(S, S')$, i.e. the subset of S that is free of norm instances that are weaker than some instance in S'. Then the following holds:*

1. $\mathbb{WF}(S, \delta) \subseteq (\delta \oplus^* S)$

2. $S \cap (\delta \ominus^* S) = \emptyset$

3. *if* $S \subseteq \delta$ *then* $\delta = (\delta \ominus^* S) \oplus^* S$

Sketch of Proof. *(1) The $Up^{\succ}$ function is constructed such that all norm instances from S are added except those that are weaker than some norm in δ.*

(2) Note that in this case the update function $Up^{\succ}$ will not add any norm instances. Rule ic^* assures that exactly the set S is removed from δ.

(3) Note that because $S \subseteq \delta$ and δ is non-conflicting by definition, no norm instance from S is weaker nor stronger than any norm instance in δ. Consequently, $Up^{\succ}(\delta, S) = \delta \cup S$.

As can be seen from above proposition, the success of addition only holds for norms that are not weaker than some norm in δ. An alternative strategy could be used that implements a more cautious approach of updating the norm instances, viz. one that guarantees that a norm change rule is executed only if no norm instance it will assert is overruled by a stronger norm instance in the original set.

Proposition 6.16 *Let $r1 = \beta \Rightarrow_* [ns_0, ..., ns_n][ns'_0, ..., ns'_n]$ be an instance-revising sc-rule s.t. all norm schemes are not* `nil`*, and $(\delta', \Delta') = (\delta, \Delta) \circledast^* tail(r1)$. Then the following holds:*

- if $inst(ns_j, \theta) \in \delta$ with $0 \leq j \leq n$ for substitution θ and $Sub^{\succ}(\{inst(ns'_j, \theta)\}, \delta) = \emptyset$ then $inst(ns_j, \theta) \notin \delta'$ and $inst(ns'_j, \theta) \in \delta'$

Sketch of Proof. *Based on a similar reasoning as proposition 6.8 item (3). Note that without the restriction that no instance in δ is stronger than $inst(ns'_j, \theta)$, $inst(ns'_j, \theta)$ might not be added. Also note that this restriction is actually too strong (for the sake of readability) as instances of each ns_j are still removed.*

With present conflict-avoiding semantics a recovery postulate for the instance-revising sc-rules (cf. 6.8 item (4)) requires strong conditions (ruling out virtually all possibilities of precedence between the norm instances), because norm instances may be removed when more preferred norm instances are added. Our mechanism does not support a way for re-asserting these removed norm instances in the process of re-instantiation. This is left for future research.

6.6 Related Work

The issue of norm change has been studied mostly from a theoretical perspective. Up to date, computational frameworks of norm change are scarce and have appeared in literature only recently. In this section we briefly compare our work to related work on the subject of norm change, thereby primarily focussing on the research on computational mechanisms rather than theoretical research. An exception is the theoretical work on norm change of Boella and Van der Torre (2004) that shows close similarities to our work. They propose a logical framework for modeling a normative system ("normative agent" as they call it) in which "count-as rules" specify when and how the norms of the system may be changed. These count-as rules resemble our norm change rules; the antecedent of the rules ranges over brute and institutional facts specifying when the norms may be changed, whereas the consequent contains actions that define how the norms should be modified.

Artikis (2009) has presented an infrastructure allowing agents to modify a protocol (a set of laws) at runtime. A protocol specification is stratified in n layers, in which layer 0 defines the domain protocol and each level $0 < k \leq n$ defines a meta protocol specifying the regulations for changing the level $k-1$ protocol. Part of a protocol specification are the "Degrees of Freedom" which define the protocol's specification components that may be modified, for example, different alternatives for a law that are replaceable at runtime. Unlike Artikis we do not distinguish between different protocol levels; under which circumstances the norms in our framework may change is statically specified by the norm change rules' conditions. Another difference is related to the principle of enforcement-indepency (cf. section 2). In Artikis' approach the changes that can be made to the protocols are hardwired in the protocols themselves, whereas in our approach the code that defines how the norms may evolve is completely separated from their specification.

In (Bou et al., 2006; Campos et al., 2009) Bou et al. and Campos et al. have proposed an approach in which the normative framework ("electronic institution" in their terminology) can change the norms at runtime. They extend a normative framework with a set of values modeling information about environment and agents, and a set of quantitative goals denoting the desired values and a transition function that specifies how the norms evolve based on the institutional goals and observed properties. The main aim of (Bou et al., 2006) is to learn the transition function that best accomplishes the institutional goals, whereas more close to our approach in (Campos et al., 2009) it is assumed that this transition function is defined by the programmer. Contrasting our approach in (Bou et al., 2006; Campos et al., 2009) changing the norms is limited to the modification of existing norms. That is, neither new norms can be added nor can existing norms be removed. Moreover, we adopt a qualitative rather than a quantitative approach in modeling information about agents and environment.

Recently, Oren *et al.* (2010) have developed a solution in which powers are used to create, delete and modify norms. Resembling our change rules, a power consist of a condition describing when it is applicable, a set of norms that will be removed and a set of norms that will be added when the power is applied. Similar to our work, a distinction is made between norm schemes (abstract norms in their terminology) and instantiated norms; powers can be used to modify both. Differently from our approach, however, does not specify what needs to occur to instantiated norms when their underlying abstract norm changes. Another difference with our work is their notion of norm subsumption, which can be used to identify families of norms that are affected by a modification rather than individual norms only. Moreover, Oren *et al.* allow for the runtime modification of powers, whereas our change rules cannot be changed at runtime.

Also related is the work of Vasconcelos *et al.* (2009) who propose a mechanism for avoiding normative conflict, albeit not in the context of a norm change mechanism as we do. Their norms take on the form of obligations, prohibitions and permissions ranging over atomic first-order formulae (denoting actions rather than declarative descriptions of a state as in our approach). They associate constraints to these formulae imposing restrictions on the domain of the variables occurring in them. In their view, a conflict occurs when some action is simultaneously prohibited and obliged/permitted and their variables are overlapping. Similar to our work, a precedence relation is introduced defining under which restrictions which norm is more preferred in case of conflict. Their mechanism of conflict avoidance then boils down to sharpen the restrictions associated to the norm that takes precedence such that the overlap is removed. A difference with our work is that we do not commit to a particular definition of when norms are in conflict. On the one hand, by annotating norms with restrictions on their variables conflicts are specified on a finer-grained level than in our approach in which conflicts can only be specified on the level of norm instances. On the other hand, our approach does not rely on such detailed information to be specified by the programmer.

Finally, and most importantly, differently from all computational frameworks mentioned above, our approach is underpinned by an operational semantics, contributing to a rigid understanding of the constructs introduced and allowing us to formally prove some basic, yet essential properties our framework exhibits. In the practical approaches mentioned here no such formal claims are given.

6.7 Conclusion and Future Work

We presented the syntax and operational semantics of generic programming constructs to facilitate the runtime modification of norms. The operational semantics is already close to the implementation of an interpreter and allowed us to mathematically investigate some basic properties our framework exhibits. We introduced rule-based constructs for modifying 1) conditional obligations and prohibitions (normative specification), 2) the detached obligations and prohibitions (deontic instances) they create, and 3) the normative specification such that also their associated deontic instances are automatically updated. The architecture we presented enables a programmer to specify when and how the norms may be changed by external agents or by the normative framework itself. Moreover, we investigated a mechanism for avoiding conflicts between norm instances.

Computational norm change is still in its infancy and we see many directions for future research. To start with, the norm change rules are currently defined at design time, and cannot be changed at runtime. Constructs for modifying norm change rules at runtime should be investigated. Further, presently, no order is defined in which the norm change rules are applied. Observe that applying norm change rules in different order might yield different results. We envisage different strategies for applying norm change rules, e.g. an explicit priority ordering among the norm change rules. This is left for future research. Moreover, we did not say anything about preserving the coherency between norm schemes (cf. (Hansen et al., 2007)). Hereto, the mechanism for avoiding normative conflicts might be used as a starting point. Also different mechanisms for avoiding normative conflict should be investigated, e.g. by enriching the precedence relation with a condition stating the circumstances under which some instance precedes another.

Chapter 7

Conclusion

Research on developing (BDI-oriented) agents has progressed rather independently from research on the development of multi-agent systems in which multiple agents interact. This has led to a gap between agent-oriented programming languages for the development of individual agents and organization-oriented programming languages for the development of coordination infrastructures. These coordination infrastructures aim to achieve and maintain the overall system's objectives by supporting individual agents to achieve their goals and preventing them from exhibiting undesired behavior. The main aim of this thesis was to develop an organization-oriented programming language that more closely accords with the concepts and constructs used for programming the individual agents that need to be coordinated.

7.1 Main Contributions

The main contribution of this thesis is a programming language by which exogenous coordination entities can be programmed in terms of organizational concepts. The representation and meaning of these concepts are chosen to match the constructs and characteristics of agent-oriented programming languages as much as possible. To gain a thorough understanding of the programming constructs proposed, their meaning is explained by a structured operational semantics. To evaluate to what extent we attained our primary research goal, let us briefly recall the main contributions and results presented throughout this thesis.

- We started (in chapter 2) with a critical analysis of related approaches to engineering organization-oriented frameworks that have appeared in the literature. To this end we identified the main characteristics and core ingredients of agents and multi-agent systems and compared them with the key properties of existing organizational frameworks. We observed that the concepts of belief, plan, and goal form the cornerstones of agent-oriented programming, whereas the cornerstones of organization-oriented programming are formed by the notions of role and norm. In evaluating existing work on organizational frameworks we also studied some literature from related fields (e.g. deontic logic and computer science) to gain a better understanding of the concept of role and norm. This analysis supported our claim that there indeed exists a gap between the construction of agents and multi-agent systems on the one hand and organizational frameworks on the other. Moreover, our analysis led to the identification of some of the most important points that should be addressed to reduce this gap. These points are discussed in the rest of this list of this contributions in the same order they appeared in chapter 2.

1. Syntax and semantics are the two pillars every programming language rests on. In agent-oriented programming the meaning of the syntactical programming constructs is often explained by a structured operational semantics, giving a precise specification of the execution of the programming constructs without suffering from the ambiguity of natural language. This formal semantics has provided a basis for 1) comparing different language constructs (possibly from different languages); 2) establishing a relation with BDI logics and; 3) formally verifying and model checking agent programs. A central theme of this dissertation has been the structured operational semantics by which the meaning of all the organization-oriented programming constructs that are proposed throughout this dissertation are explained. The result is a unambiguous description of the meaning of the programming language constructs. The operational semantics allowed us to make some formal claims about the key properties our framework exhibits and paved the road for some preliminary work on model checking (Dennis et al., 2009). To our best knowledge, we are the first to present the full operational semantics of an organization-oriented programming language.

2. The environment the agents are situated in has been recognized a first-class engineering abstraction with distinguished responsibilities. One of the key responsibilities of the environment is to encapsulate resources and functionality that the agents may exploit to reach their objectives. Most existing organizational frameworks do not fulfill this responsibility, but rather focus on the ability to coordinate and structure the multi-agent system – another key responsibility assigned to the environment the agents are situated in. In chapter 3 we introduced the notion of organizational artifact, a computational entity that encapsulates resources and provides functionality the agents may use in achieving their objectives. Organizational artifacts are compositional in the sense that they can perform actions upon other artifacts. The internal architecture of an organizational artifact is modular in the sense that more sophisticated organizational constructs for coordinating and structuring a multi-agent system (cf. roles, norms, sanctions and

norm change rules) can be matched and mix depending on specific needs. These constructs can then be glued together by means of the coordination language we described in chapter 3.

3. In a multi-agent system that is characterized by a degree of openness, agents that are unknown to the organizational artifact's designer may dynamically start their interactions. To guide these agents in interacting with the organizational artifact in a meaningful way, and to prevent the artifact from ending up in undesired situations, we enriched the artifact with a normative dimension (chapter 4). We argued that a declarative view on norms in which the norms relate to descriptions of situations that should be achieved or avoided has several benefits over a procedural view on norms in which the norms range over actions the agents must (not) perform. The most important benefits being that a declarative view better respects the agent's autonomy and more closely accords with the concept of achievement goal that is typically found in agent programming languages. The normative dimension we proposed is comprised of conditional norm schemes that instantiate declarative obligations and prohibitions and sanctioning rules for motivating the agents to abide by the norms. The presence of a formal operational semantics allowed us to demonstrate some essential properties of the behavior of our norms.

4. If unknown agents dynamically start their interactions with an organizational artifact, at design-time it is hard to predict the exact nature of their demeanor. The set of norms that has been put in place at design-time might need adjustment at run-time to better cope with the actuality of the running system. Research on how a set of norms may evolve at run-time in the context of a computational enforcement mechanism is few and has only appeared recently. Existing research on this matter does not investigate how the invoked obligations and prohibitions evolve when their underlying norm scheme's are modified. In chapter 6 we proposed rule-based constructs for changing the norms at run-time, also taking into account how already existing obligations and prohibitions evolve when their underlying norm schemes are altered. Another distinguishing feature of our norm-change rules is that both agents and organizational artifact are entitled to initiate norm change.

   If norms change at run-time it becomes increasingly difficult to foresee possible normative conflicts, loosely meaning that a set of norms prescribes a behavior that will inevitably lead to a violation. Therefore, we enriched our norm change mechanism with means for avoiding normative conflicts. Because there is no consensus on whether normative conflicts should always be avoided and a definition for normative conflict is hard to come up with (as argued in chapter 4), our mechanism relies upon the presence of a relation indicating which norm instances are conflicting. This relation can be either provided by the system's designer or can be automatically generated based on some definition of normative conflict.

5. Roles are the cornerstones of agent organizations. They are typically used as placeholders for agents to allow the system's designer to abstract away from the individuals that will play them. By means of an extensive analysis of existing literature on roles, in this thesis we have shown that roles are really more than merely placeholders for agents. We identified four key characteristics roles should exhibit to reduce the gap between agent development and organization development. That is, roles must be 1) organization-centric meaning that they are defined as an intrinsic part of the organizational artifact; 2) employable, meaning that tasks can be delegated to roles; 3) prescriptive, which means that roles may guide their players in the process of interacting with the artifact, and 4) BDI-based meaning that they should be developed by employing the same concepts as agents. In chapter 5 we developed a solution exhibiting all these properties, in which a role can be seen as the organizational counterpart of an agent acting as the interface through which agents interact with the organizational artifact. This has not been done before and is an important step in reducing the gap between agent-oriented approaches and organization-oriented approaches.

6. We investigated the possibility to use norms for expressing various constraints related to roles in chapter 5. This novel approach opens the possibility to express these organizational constraints not only as hard-constraints that are not violable, but also as soft-constraints that can be deviated from. This increases flexibility and better respects the agents' autonomy. Moreover, this investigation has led to some directions for possible extensions of the norm scheme's expressiveness.

7. During its lifetime, an agent is faced with choices about what to do next, e.g. executing a plan to reach a goal or updating beliefs on the basis of incoming percepts. The best order (if any) in which agents perform such activities is application dependent and is encoded by their deliberation cycle. Likewise, different strategies exist for an organizational artifact to perform its activities, e.g. enforcing all norms versus enforcing only some in time critical systems. Inspired by the notion of a deliberation cycle we introduced the concept of coordination cycle (in chapter 3), by which the organizational artifact determines what to do next. We defined operational semantics in such a way that there is a clean separation between the object level pertaining to the meaning of the programming constructs and the meta level pertaining to the order in which the constructs are applied.

7.2 Ten Things to Think About Before Finishing This Thesis

We believe that the results of this thesis are a significant step towards reducing the gap between agent-oriented and organization-oriented approaches for developing multi-agent systems. We also believe that there is still much work to be done to flexibly build organization-oriented multi-agent systems in which BDI-oriented agents interact. Indeed, the gap is not completely closed and still much research needs to be done, both on the side of organization-oriented programming (as we did in this thesis), but also on the side of agent-oriented programming. On the way to bridge the gap we also identified many interesting points for future research that not directly contribute to reducing the gap. In the individual chapters we have occasionally discussed lines for future research. Here, we will capture their essence and point out directions we have not mentioned before. These are the ten things to think about before finishing this thesis.

1. In this thesis we offered various programming constructs and concepts for building organization-oriented entities that better accord with the constructs agents are built with. However, we did not say anything about how these concepts may influence the agents' decisions. To fully exploit the use of organizational abstractions in the design of multi-agent systems it is of crucial importance that agents are somehow able to understand and reason about the concepts and constructs offered by an organizational artifact in making their decisions. Enabling them to do so, requires extended capabilities of the agents who need to reason with organizational constructs, but also requires additional mechanisms for the organization to make agents aware of relevant organizational constructs (e.g. what roles can be played and which capabilities are expected from their player, and which obligations are directed to them). We believe that we should be conservative in extending the agents' capabilities, as this would limit the types of agents that can interact with an organizational artifact. We think that the roles presented in chapter 5 are a good candidate for fulfilling a bridging function between agent and organizational artifact by extending their capabilities to, for example, translate obligations to goals that can be communicated to their player. We deem this research direction to be the most important one in bridging the gap between agent-oriented software engineering and organization-oriented software engineering. Yet, it is the direction in which most work remains to be done. For a more elaborate discussion on this matter the interested reader may refer to the work of Van Riemsdijk *et al.* (2009).

2. In chapter 4 we have argued for a declarative view on norms, meaning that norms refer to a declarative description of a situation that must (not) be established. Although this has several advantages, we believe that extending our norms to also include procedural norms for enforcing or preventing the execution of actions might be of added value. For example, when we are interested in choreographing the concrete steps an agent must take in reaching a more abstract situation, declarative norms may fall short. Combining both declarative and procedural norms in one framework facilitates

the enforcement of *states* to be reached as well as the *means* to reach them. It is important to emphasize that this exercise is not simply a matter of defining the operational semantics for the norms of one of the existing normative frameworks. The norms in existing work typically range over one single action; we envisage an integration of more expressive action patterns, as can, for example, be found in the work of Meyer (1987). Also the work of Astefanoei *et al.* (2010) in which declarative counts-as statements are combined with procedural norms in the form of choreography expressions may serve as a good basis for this extension.

3. The norms presented in chapter 4 have the purpose of regulating the agents' behavior; they are regulative norms. It has been recognized that in specifying normative systems, besides regulative norms, also constitutive norms are needed for connecting more concrete brute facts with more abstract institutional facts. Such constitutive norms facilitate to specify regulative norms on a higher level of abstraction. Suppose, for example, we are interested in expressing an obligation to have paid the conference fee. Then constitutive norms allow us to specify which different methods for paying (e.g. paying by credit card, issuing a check, paying in cash) count as a payment, such that in specifying the regulative norm we can refer to the more abstract notion of payment. We believe that the counts-as rules we presented in (Dastani et al., 2008, 2009) might serve as a basis for an integration with our regulative norms.

4. The influence of norm schemes has been restricted to the organizational artifact they are part of. That is to say, a norm scheme can only refer to (brute or institutional) facts of the artifact it belongs to and can only instantiate obligations and prohibitions in that artifact. This characteristic limits the ability to build truly distributed systems. Suppose, for example, we would break up our reviewing system in different organizational artifacts each implementing the reviewing functionalities of a conference sub-track. How to express a norm scheme specifying that a reviewer can only be on the program committee for one sub-track or a norm scheme that specifies that not more than a certain amount of papers may be accepted for the whole conference remains to be explored. To express such norms, a norm scheme must refer to the state of more than one organizational artifact. Research on the distribution of norms can be found in (Gaertner et al., 2007) in which it is investigated how to manage (procedural) norms in an architecture in which the activities are distributed, but the problem of how to express (declarative) norms that span across multiple artifacts is not addressed.

5. The issues of how to define and deal with normative conflicts and normative incoherency are known as long-standing problems in the field of deontic logic (Hansen et al., 2007). Also in our computational norm enforcement mechanism these turned out to be difficult matters. No definite answers are

provided on this subject. We think it will be interesting to investigate to what extent model checking techniques can be used to identify situations in which, say, violations cannot be avoided. Such an analysis not only reveals possible conflicts and incoherences given a specific program, but might also give new insights in when norm instances are in conflict and norm schemes are incoherent. Moreover, as model checking is typically done off-line, no computationally expensive mechanisms are needed for detecting conflicts at run-time.

6. A mechanism to prevent normative conflicts from happening has been investigated in the context of our norm change mechanism in chapter 6. As we argued, multiple strategies exist for avoiding normative conflicts and we only presented one. More research in this direction is needed. Moreover, preventing normative conflicts in the context of the norm enforcement mechanism is equally important. An extension of the enforcement mechanism to avoid normative conflicts is left for future research. Moreover, also attention should be given to a mechanism for preventing normative incoherency.

7. Roles are an important ingredient in structuring organizational artifacts. In chapter 5 we explored how various constraints on roles can be expressed by means of our norms presented in chapter 4. This exploration revealed that the norm schemes are not expressive enough to model some constraints. It will be interesting to explore this line of research further, as this might lead to the suggestions for extending the expressiveness of the norm schemes. Moreover, we remark that also other means (than constraints) for structuring the roles of an organizational artifact need to be investigated. In (Grossi et al., 2005) Grossi *et al.* for example, formally analyze some fundamental structural relationships that may exist between roles (viz. power, coordination and control). But we can also think, for example, of inheritance of roles and letting roles enact other roles (cf. (Kristensen, 1995; Colman and Han, 2007)). As observed in chapter 5 letting roles enact roles of other artifacts might be a useful means for composing different organizational artifacts.

8. In this dissertation we have given most attention to the concepts of norm and role as these are arguably the key organizational concepts in constructing organization-oriented agent systems. However, also a wide range of other organizational concepts has been proposed to date. Commitments (cf. (Torroni et al., 2009)), task hierarchies (cf. (Tambe and Zhang, 1998)) and permissive norms (cf. (Hansen et al., 2007)) are just but a few examples. No matter how useful a concept might seem, before our organizational framework is extended to include it, we stress that the relation of that concept with the concepts proposed in this thesis first needs to be explicated. We believe that our operational semantics provides a profound basis for such an analysis.

9. One of the benefits of an operational semantics is that it paves the road for model checking activities (cf. (Baier and Katoen, 2008)). Indeed, the operational semantics facilitated the automated formal verification of organization-oriented programs written in our counts-as based organizational language (see chapter 2) that formed the basis of present organization-oriented programming language, e.g. (Aştefănoaei et al., 2008; Dennis et al., 2009). This line of research should be continued to enable formal verification of programs written by the programming constructs proposed throughout this thesis.

10. Defining the meaning of the programming constructs in a rigid manner and proving that some desirable properties are exhibited, only provides a partial answer to the question about the rightness of our programming language. Indeed, as we pointed out in the introduction, the raison d'être of a programming language is largely determined by its pragmatics, i.e. the way in which the language is intended to be used in practice (Watt, 2004). The question of how pragmatic a programming language actually is, is a difficult one to answer. The answer to that question is not simply yes or no; it is either useful or useless and all shades in between. To at the very least make some claims about its pragmatics we deem an implementation to experiment with the different constructs indispensable. This way our proposal can be put to the test and provides means to identify useful coordination strategies. Our operational semantics is already close to the implementation of an interpreter and at this moment we have already taken some first steps towards a prototype implementation. A discussion of this prototype and the lessons learned can be found in appendix A.

As above list of future directions shows, this is not the final word on the organization-oriented development of agent systems. Quite the contrary is true. The above points are not just meant to think about before finishing this thesis; they are meant to be born in mind after having finished this thesis as well. Of course, this list is far from being exhaustive. Indeed, we hope our work will inspire many researchers to carry on this line of research and make the multi-agent systems paradigm to live up to its promise as a suitable paradigm for developing complex, distributed software.

Appendix A

A Prototype Implementation

Throughout this thesis we have stressed the importance of an implementation. Here we briefly comment on a prototype we have been experimenting with and show how the prototype was used to implement a simple application involving an auction. This prototype is mainly based on the normative constructs of chapter 4 and the roles of chapter 5, but does not cover everything. It should be emphasized that, although we developed our prototype with the operational semantics in mind, no investigation has been done to verify that the prototype is indeed faithful to the semantics. At the very least we can say that we implemented a prototype based on the ideas presented in this thesis. The main goal of this prototype is to provide the opportunity to already experiment with some of the language's features in an early phase. On the one hand this effort shows a possible road towards an implementation. On the other hand experimenting with some of the language's features might lead to new insights on possible extensions of the operational semantics.

A.1 General Architecture

The general architecture of our prototype is shown in figure A.1. As can be seen in this figure, the prototype consists of two parts. One part is responsible for simulating the roles and positions, whereas the other part is responsible for maintaining the brute state, institutional state and norm instances, and the execution of effect rules, norm schemes, etc.

The part of the organizational artifact that is responsible for the latter is handled by the Jess Norm Engine (JNE). As its name suggests, the JNE is based

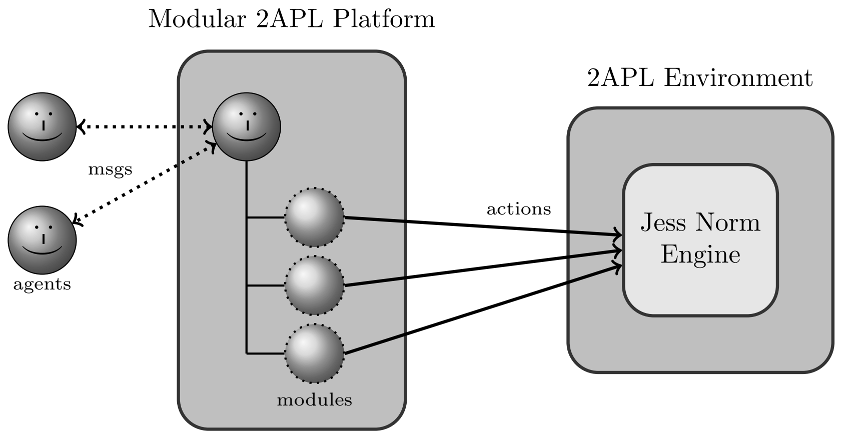


Figure A.1: The general architecture of our prototype. The prototype consists of two parts, the Jess Norm Engine (JNE) and an extension of the 2APL platform. The JNE that is responsible for maintaining brute facts, institutional facts and norm instances, and executing effect rules, norm schemes and sanctioning rules. Positions are simulated by 2APL modules that are created and manipulated by a special designated manager agent running on the 2APL platform. External agents (shown outside of the platform) interact with their positions via the manager agent.

on the Jess framework.[1] Jess (acronym for Java Expert System Shell) is a rule engine and scripting environment that is implemented in the Java programming language. Jess facilitates to straightforwardly implement computationally efficient rule based systems. That is, software implemented by means of declarative rules that are used to reason about knowledge as provided by the system developer. One of the most attractive features of Jess is its seamless integration with Java that gives the developer access to the entire Java API from its scripting environment and access from a Java program to the API of the Jess engine. The knowledge of the Jess engine is represented as a set of facts that is stored in the engine's working memory. The core ingredient of the Jess rule engine are the rules that define which actions should be performed based on a certain state of the working memory. The rules of the Jess engine are forward-chaining-rules of the form `C => A` in which `C` is a condition referring to facts as present in the working memory and `A` are the actions to be performed in case `C` holds. Note the similarity of these rules with the traditional `if C then A` statements of a procedural language. Examples of actions that can be used in the right-hand-side of a rule are the assertion, modification and deletion of facts in working memory. Rules (and also facts) can be stored in modules. When running Jess only one module will have

[1]http://www.jessrules.com

focus at a time and only the rules belonging to that module will be considered in firing them.

The roles and positions are simulated on an extension of the 2APL framework. This version of the 2APL platform involves an extended version of the modules that are used in 2APL and has been described in (Cap, 2010). 2APL modules are built of the same mental state as a 2APL agent. Agents (and modules) can dynamically create modules and manipulate their goals and beliefs, and activate and de-activate them. In our prototype the modules are used to simulate positions. The external agents interact with the positions they play via a designated `manager` agent who manages all the positions (modules). Note that this is differently from how it has been proposed in chapter 5 in which the agents interact directly with the organizational artifact and their positions. Because the JNE is wrapped by a 2APL external environment, the modules (and agents) can directly perform their external actions upon the JNE. The 2APL platform thus acts as the interface between the external agents and the organizational artifact as implemented by the JNE.[2] In what follows, we explain in some more detail how to implement organizational artifacts with roles in our prototype.

We illustrate how our prototype can be used by showing the implementation of an example involving a simplified auction artifact. Products are put on auction by an agent playing the role of auctioneer. Agents playing the buyer role are informed about products that are put online by the auctioneer. After having informed all potentially interested buyers, the auctioneer starts the auction by announcing the first round. The auction consists of different rounds in which buyers can submit a bid. At the end of each round the auctioneer collects all bids, and informs all buyers about the highest bid of that round. A buyer wins the auction when it has submitted the highest bid in a round and is not overbid by any agent in the next round. The winner of the auction is expected to pay the price equal to its last bid. Only one product is put on auction at the same time.

A.2 Organizational Artifacts Simulated in Jess

To implement an organizational artifact (without positions) in our prototype is to translate the program specifying the artifact into executable Jess code that is handled by the dedicated JNE framework. The syntax provided by Jess differs from the syntax proposed throughout this thesis. Because the implementation here is merely a prototype, no parser has been implemented to translate the code directly. The disadvantage is that one has to be careful not to use constructs that are not supported by our original framework whilst translating the code. The advantage is that one can easily deploy Jess functionality to experiment with possible extensions of the original language. How the various programming constructs are translated to executable Jess code is shown in figure A.2.

[2]In the current implementation the external agents run on the same platform as the modules, but could in principle run on a different platform.

```
Facts:
 fact_1
 ...
 fact_n
```

$\mapsto$

```
(deffacts brute-facts
 (fact_1)
 ...
 (fact_n)
)
```

(a) Translation of facts.

```
{fact_1,...,fact_n}
 action(term_1,...,term_k)

{fact_1,...,fact_m}
```

$\mapsto$

```
(defrule E::action
 (fact_1) ...  (fact_n)
 (DO::action term_1,...,term_k)
=>
 (α fact_1) ...  (α fact_m)
)
```

(b) Translation of effect rules.

```
label(term_1,...,term_k):
< fact_1 and ...  and fact_n
, M(fact_1 and ...  and fact_m)
, (fact_1 and ...  and fact_k)
>
```

$\mapsto$

```
(norm
 (label term_1,...,term_k)
 (and fact_1 ...  fact_n)
 (M (and fact_1 ...  fact_m))
 (and fact_1 ...  fact_k)
)
```

(c) Translation of norm schemes.

```
fact_1 and ...  and fact_n
 =>
{fact_1, ...  , fact_m}
```

$\mapsto$

```
(defrule S::rulename
 (fact_1) ...  (fact_n)
=>
 (α fact_1) ...  (α fact_m)
)
```

(d) Translation of sanction rules.

Figure A.2: Translation of the programming constructs for specifying an organizational artifact into executable Jess code. Each diagram shows the programming constructs from our original language on the left and the corresponding Jess code on the right. The special designated Jess actions for the assertion and deletion of facts are denoted by α.

Recall that the domain specific state of an artifact is described by a set of brute facts and that the information about which roles are enacted and which norms have been violated or fulfilled is stored by the institutional facts. In Jess these facts are straightforwardly translated to Jess' facts. To distinguish between the two, brute facts are stored in a module `B`, whereas the institutional facts are stored in a module `I`. The initial state of the brute state is specified by stating the facts that hold from the start. Specifying the initial state in Jess boils down to providing all the brute facts that should initially hold as argument to the built-in `deffacts` function. This is depicted in figure A.2a. The fact that the auction artifact initially starts at round 0 is expressed as follows:

```
(deffacts brute−facts
        (B::rnd 0)
)
```

The `B::` preceding the predicate name `rnd` is for indicating that this fact belongs to the module encompassing the brute facts. It should be noted that Jess facts can take on the form of ordered and unordered facts. Ordered facts are Jess' implementation of plain lists consisting of a head that usually acts as the category for the fact and the tail as its fields. The above fact is an example of an ordered fact named `rnd` having one element, viz. `0`. The elements of unordered facts, as the name already suggests, are not ordered and their fields are distinguished by naming them. They need to be defined by a template before they can be used in a program. Consider, for example, the template definition of a bid:

```
(deftemplate B::bid
  "A bid placed by an agent."
  (slot agent)
  (slot amount)
  (slot product)
  (slot rnd)
)
```

The effect rules specify the effects actions have on the brute state. They are translated to Jess rules that are stored in the module `E` as shown in figure A.2b. The action and its pre-condition specifying the circumstances under which it is executable is written as the left-hand side of a Jess rule. Note that the action of the effect rule is denoted by a special designated fact with the same name as that of the action and is stored in the module `DO`. The first argument of every action is the identifier of the agent performing it. The framework ensures that these special facts are derivable from the working memory when an agent has just performed the action denoted by it. The right-hand side of the rule specifies the modifications that should be made to the brute state when the rule fires, i.e. when the action is performed and its pre-condition holds. Consider, for example, the following effect rule:

```
(defrule E::bid
  (DO::bid ?a ?x ?p ?r)
  (B::rnd ?r)
  =>
  (assert (B::bid (agent ?a) (amount ?x) (product ?p) (rnd ?r)))
)
```

specifying that an agent `?a` may bid amount `?x` for product `?p` in round `?r` if the current round is `?r`. After a succesful performance a corresponding bid fact will be asserted to the brute state.

The norm schemes for expressing the conditional obligations and prohibitions are implemented in Jess by using the custom defined function `norm` as shown in figure A.2c. The first argument denotes the unique label of the norm scheme. The second argument pertains to the norm scheme's condition taking on the form of a conjunction of facts. The obligation or prohibition is denoted by the third argument, in which modality `M` can be either `O` in case of an obligation or `F` in case of a forbiddance. Finally, the fourth argument denotes the norm scheme's deadline. An example of a norm scheme is the following `pay` norm:

```
(norm
  (pay1 ?a ?x)
  (B::winner (agent ?a) (amount ?x))
```

```
    (O (B::paid ?x))
    (B::rnd 0)
  )
```

stating that the winner of the auction is obliged to pay the amount equal to its highest bid. The winner is given time to do so, before the auctioneer resets the system to round 0 again. Recall that a violation of this norm is indicated by the assertion of a violation fact. If the winner does not respect this obligation a new obligation is issued to pay the amount plus a 25 euro fine, as expressed by the following norm scheme:

```
  (norm
    (pay2 ?a ?amount)
    (and  (I::viol pay1 ?a ?amount)
    )
    (O (B::paid (+ ?amount 25)))
    (B::forever)
  )
```

Note that the winner is given ad infinitum to fulfill this obligation. The reason is explained by the sanctioning rules.

The sanction rules by which we define sanctions that should be imposed on the brute state of the organization given a certain normative judgment are directly translated to Jess rules. This is shown in figure A.2d. The left-hand side of these rules refers to brute and institutional facts and the right-hand side specifies which brute facts are to be added to or retracted from the brute state. The sanction rules are defined in a module named `S`. Consider, for example, the following sanction rules:

```
  (defrule S::pay1
    (I::viol pay1 ?a ?x)
    (I::rea ?agnt "buyer" ?a)
    =>
    (assert (B::blacklist ?agnt))
  )

  (defrule S::pay2
    (I::obey pay2 ?a ?x)
    ?bl <- (B::blacklist ?agnt)
    =>
    (retract ?bl)
  )
```

The first sanction rule specifies that an agent who violates the payment norm will be blacklisted. Observe how the `rea` fact is used to obtain the identity of the agent playing the buyer role via its position. An agent who is blacklisted can only be get of the list again by complying with norm scheme `pay2`, i.e. by paying the amount of its winning bid plus the fine. This is expressed by the second sanctioning rule. Information about blacklisted agents can be used to refuse agents to enact the buyer role in the future.

The JNE implements a fixed coordination strategy in which all the norm schemes are triggered, all norm instances are checked for compliance and all applicable sanction rules are applied. This pertains to a totalitarian strategy as discussed in chapter 4. A regimentation strategy has not been implemented.

Often, an agent that performs an action is interested in the result. Indeed, in 2APL it is not atypical for an external action to return a result. This feature has been overlooked in the operational semantics and we discovered it when implementing the prototype. Returning a value (a term) from an action that can be perceived by the agent performing it is achieved by means of asserting a ground fact named `result` in an effect specification. The framework ensures that this fact is bound to the result of the return parameter of the external action.

A.3 Roles Simulated in Modular 2APL

In chapter 5 we did not commit to a specific agent-oriented language for implementing roles. For the example, however, used in chapter 5 we committed to the 2APL programming language. For the implementation of our prototype we did the same. In what follows we briefly explain how the constructs that were introduced in chapter 5 are simulated by our prototype by showing some examples from our auction artifact.

In our prototype 2APL modules are used to simulate roles. In what follows we call everything a role, although in the underlying model it is really a module. To enact a role an agent sends a message to the manager agent, e.g. by sending a message:

```
send(manager, request, enact(buyer,[bike,160,30]))
```

an agent indicates it wishes to enact the buyer role for buying a bike for at most 160 euros and indicates that it likes to increase its bid by 30 euros each round. The manager role then instantiates the buyer role and tries to execute a suitable constructor for instantiating a position. Recall that the constructor of a role may query the role's mental state and organizational state. In 2APL, however, the notion of constructor has not been implemented and it is only possible to query the role's beliefs. How we circumvented this limitation is illustrated by the following example showing the implementation of the buyer's constructor:

```
enact([Product,MaxPrice,Step]) <- true |
{
  B( player(Name) );
  @org( perform( blacklist([Name]) ), R );

  if B(R = [result([no])]) then
  { SetStep(Step);
    adopta( bought(Product,MaxPrice) );
    enactSuccess( )
  }
  else
  { enactFail( )
  }
}
```

The constructor itself is implemented as a traditional PC-rule in the buyer role. Consequently, it cannot be guaranteed that no other positions perform their actions whilst executing the constructor. This is differently from the operational semantics. To simulate the constructor's pre-condition, we introduced the special designated `enactSuccess` and `enactFail` procedures to indicate the satisfaction or failure of the constructor's condition. In the above example the buyer role may be enacted when the agent that tries to enact the role is not blacklisted. We simulated this test on the brute state by introducing a special action `blacklist([Name])` that returns `yes` in case the brute fact indicating that the agent named `Name` is blacklisted can be derived and `no` otherwise. The actual constructor plan that is only executed if the position's pre-condition holds makes the buyer adopt a goal to have bought the desired product and update its beliefs with the desired step size for bidding. The above enact PC-rule thus simulates the constructor:

```
enact([Product,MaxPrice,Step])
<- B(player(Name)) & E(not blacklisted(Name)) |
{
  SetStep(Step);
  adopta( bought(Product,MaxPrice) );
}
```

When an agent has succesfully enacted a position, the manager sends the agent the position's identifier. The role's destructor is implemented in a similar fashion.

Once an agent has enacted a position and has received the position's id from the manager, an agent can activate and de-activate its position by sending the manager a request to activate/de-activate its position, e.g.:

```
send(manager, request, activate(Pos))
```

in which `Pos` is set to the identifier of the position.

Once activated the buyer position will start acting on behalf of its player, by adopting plans to achieve its goal to buy the product for the best price available (but not more than its player is willing to pay for it). The PG-rules and PC-rules that are used by the buyer role are directly implemented in 2APL and will not be explained further. In pursuit of its goal to buy the product for the best price available, the buyer will respond to messages that were sent by the auctioneer. The auctioneer, for example, publicly announces each round's highest bid and at the end of the auction informs each individual buyer position whether it has won the auction or not. A buyer position responds to a win/los declaration by forwarding the information to its player, e.g.:

```
message(A, inform, _, _, winner(Winner, Product, Price))
<- me(Winner) |
{
  send(manager, inform, won(Product, Price))
}
```

Note that the buyer position does not send its messages directly to its player. It sends the message to the manager agent which in turn forwards it to the player. More specifically, it wraps the content of the position's message (say `Content`) in its own message's content `msg(Pos,Content)` in which `Pos` is the identifier of the position sending the message. An agent who is just informed by its position about winning or losing the auction, may respond to this message as described by the following PC-rule:

```
message( manager, inform, _, _, msg(Pos,Content) ) <- true |
{ if B(Content = won(Product,Price)) then
  { send( manager, request, adoptg(Pos, paid(Product,Price)) )
  };
  if B(Content = lost(Product,Price)) then
  { send( manager, request, deact(Pos) )
  }
}
```

This agent thus delegates its position a goal to pay for the product if it has won the auction and deacts the position in case someone else won. It should be noted that in the prototype it is not possible to delegate a position any goal. In fact, which goals exactly can be adopted should be specified in the manager position. The same goes for dropping goals and adopting/dropping beliefs. The actions to inspect the mental state of a position were not implemented.

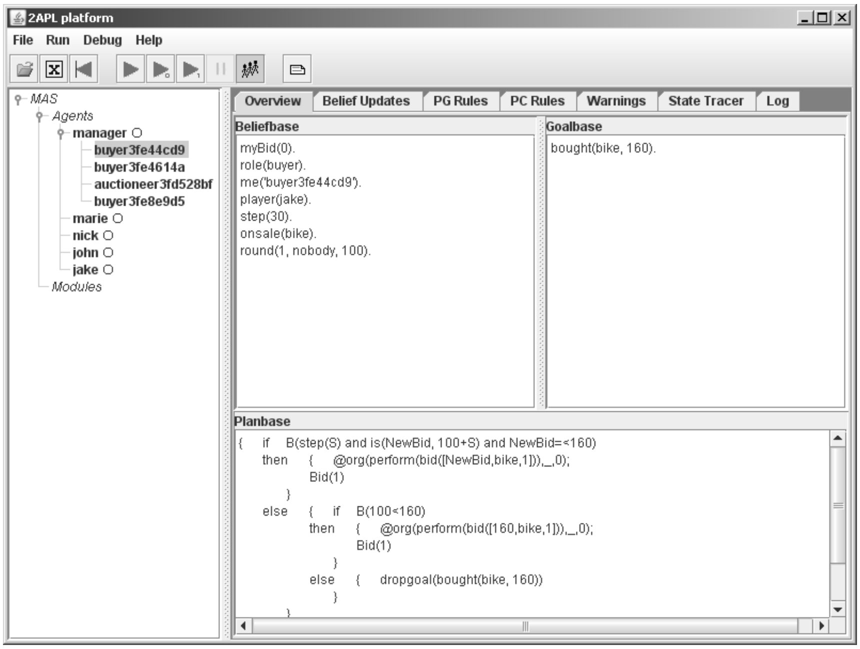
A.4 Inspecting Organizational Artifacts

When constructing and running an organizational artifact it would be helpful to have some way of inspecting the artifact's internal state. For example, one may wish to inspect the artifact's current brute and institutional state and inspect the mental state of the positions that are played by external agents. Because the positions are simulated by the 2APL platform one can use the functionality of this platform for inspecting the position's mental state and display the contents of the messages that were interchanged between positions and between positions and their players. Some screenshots showing the use of the 2APL platform for inspecting the state of positions and the messages being sent are shown in figures A.3 and A.4.

The part of the organizational artifact that is responsible for maintaining the brute and institutional state, and the active prohibitions and obligations is implemented by the JNE. To visualize this part of the artifact we implemented a simple inspector that allows one to cycle through the whole history of states, starting with the artifact's initial state. Each state pertains to the situation the artifact is in that results after the performance of an action. An example screenshot of the artifact inspector is shown in figure A.4.

A.5 Conclusion and Lessons Learned

We implemented a prototype that allowed us to experiment with some of the programming constructs that were presented throughout this thesis, roles and norms in particular. It should be noted that also other attempts have been made to implement an interpreter based on the ideas presented in this thesis. For



Beliefbase

```
myBid(0).
role(buyer).
me('buyer3fe44cd9').
player(jake).
step(30).
onsale(bike).
round(1, nobody, 100).
```

Goalbase

```
bought(bike, 160).
```

Planbase

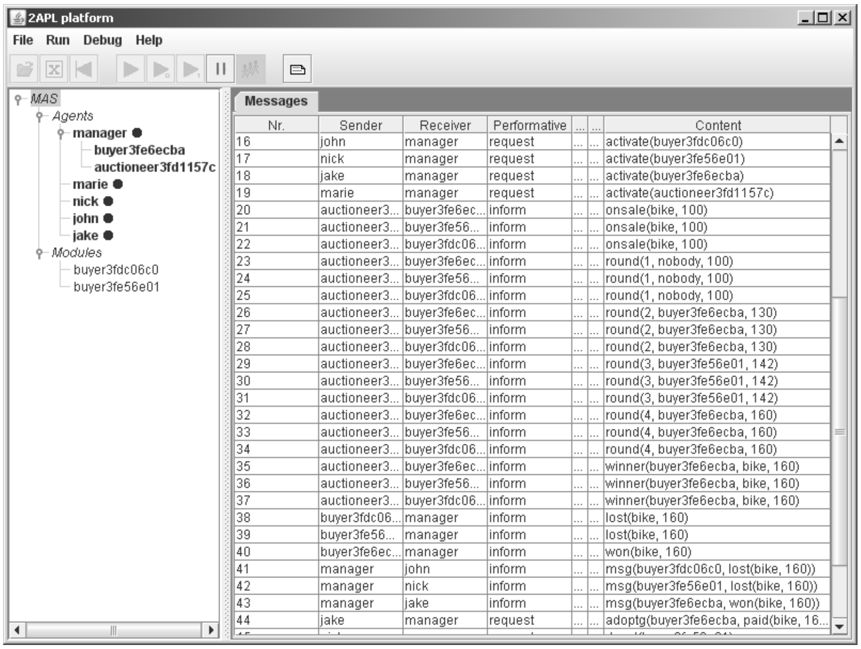
```
{   if   B(step(S) and is(NewBid, 100+S) and NewBid=<160)
    then   {   @org(perform(bid([NewBid,bike,1])),_,0);
               Bid(1)
           }
    else   {   if   B(100<160)
               then   {   @org(perform(bid([160,bike,1])),_,0);
                          Bid(1)
                      }
               else   {   dropgoal(bought(bike, 160))
                      }
           }
```

Figure A.3: Using the 2APL platform to inspect a buyer position's mental state.

example, an implementation based on the counts-as based norms can be found in (Adal, 2010).

During the implementation of this prototype together with the auction and some other examples some interesting observations have been made. These observations point to possible directions for future research. Some other observations led to revisions of earlier versions of our operational semantics. For example, in a first attempt to model a prohibition, no violation occured when an agent performed an action that simultaneously establishes the deadline and the forbidden state. (See also the discussion at the end of section 4.2 of chapter 4.) An example revealed that this did not adequately capture the intuition, which made us revise the operational semantics. In the rest of this section we briefly discuss three observations that may lead to future research directions.

The auctioneer agent is currently played by an external agent who delegates it a goal to put a product on auction. Ideally, these external agents should not enact the auctioneer role directly, but instead play a seller role that delegates products to be put on auction to the auctioneer. But then the question becomes, who plays the auctioneer role? It would be rather strange if any external agent would play this role. An agent that plays it, should at the very least be trusted by the organization. This suggests that the auctioneer is an internal agent of the



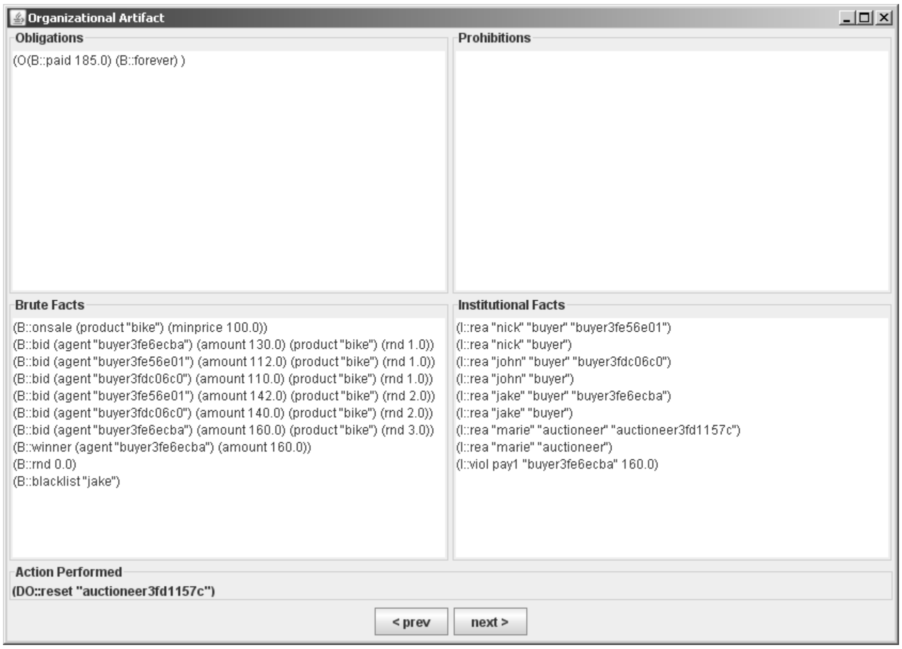
Nr.	Sender	Receiver	Performative	...	...	Content
16	john	manager	request	...	...	activate(buyer3fdc06c0)
17	nick	manager	request	...	...	activate(buyer3fe56e01)
18	jake	manager	request	...	...	activate(buyer3fe6ecba)
19	marie	manager	request	...	...	activate(auctioneer3fd1157c)
20	auctioneer3...	buyer3fe6ec...	inform	...	...	onsale(bike, 100)
21	auctioneer3...	buyer3fe56...	inform	...	...	onsale(bike, 100)
22	auctioneer3...	buyer3fdc06...	inform	...	...	onsale(bike, 100)
23	auctioneer3...	buyer3fe6ec...	inform	...	...	round(1, nobody, 100)
24	auctioneer3...	buyer3fe56...	inform	...	...	round(1, nobody, 100)
25	auctioneer3...	buyer3fdc06...	inform	...	...	round(1, nobody, 100)
26	auctioneer3...	buyer3fe6ec...	inform	...	...	round(2, buyer3fe6ecba, 130)
27	auctioneer3...	buyer3fe56...	inform	...	...	round(2, buyer3fe6ecba, 130)
28	auctioneer3...	buyer3fdc06...	inform	...	...	round(2, buyer3fe6ecba, 130)
29	auctioneer3...	buyer3fe6ec...	inform	...	...	round(3, buyer3fe56e01, 142)
30	auctioneer3...	buyer3fe56...	inform	...	...	round(3, buyer3fe56e01, 142)
31	auctioneer3...	buyer3fdc06...	inform	...	...	round(3, buyer3fe56e01, 142)
32	auctioneer3...	buyer3fe6ec...	inform	...	...	round(4, buyer3fe6ecba, 160)
33	auctioneer3...	buyer3fe56...	inform	...	...	round(4, buyer3fe6ecba, 160)
34	auctioneer3...	buyer3fdc06...	inform	...	...	round(4, buyer3fe6ecba, 160)
35	auctioneer3...	buyer3fe6ec...	inform	...	...	winner(buyer3fe6ecba, bike, 160)
36	auctioneer3...	buyer3fe56...	inform	...	...	winner(buyer3fe6ecba, bike, 160)
37	auctioneer3...	buyer3fdc06...	inform	...	...	winner(buyer3fe6ecba, bike, 160)
38	buyer3fdc06...	manager	inform	...	...	lost(bike, 160)
39	buyer3fe56...	manager	inform	...	...	lost(bike, 160)
40	buyer3fe6ec...	manager	inform	...	...	won(bike, 160)
41	manager	john	inform	...	...	msg(buyer3fdc06c0, lost(bike, 160))
42	manager	nick	inform	...	...	msg(buyer3fe56e01, lost(bike, 160))
43	manager	jake	inform	...	...	msg(buyer3fe6ecba, won(bike, 160))
44	jake	manager	request	...	...	adoptg(buyer3fe6ecba, paid(bike, 16...

Figure A.4: Using the 2APL platform for inspecting messages.

organization rather than a role played by some external agent.

In the current design as discussed in chapter 5 players can inspect and modify the entire mental state of their positions. During implementation we needed to change the representation of specific beliefs and goals of the buyer role more than once. These changes often led to a change in the external agents also. This made us wonder whether the complete mental state of a role should be inspectable by its player or if we should somehow provide the player a stable interface that specifies which beliefs and goals can be inspected and modified. This way the underlying implementation that may change can be hidden to the implementation according to the principle of encapsulation.

Finally, in implementing the auction and some other applications we found that the cases in which the mental state of a position needs to be inspected and modified could all be handled by means of PC-rules implemented in the role. This way it could be defined which, when and how the position's goals and beliefs can be modified/inspected rather than modifying/inspecting all of them any time as suggested in chapter 5. It thus seems that the actions to directly modify and inspect a position's mental state as presented in chapter 5 are superfluous. This topic needs further investigation.



```
(O(B::paid 185.0) (B::forever) )
```

```
(B::onsale (product "bike") (minprice 100.0))
(B::bid (agent "buyer3fe6ecba") (amount 130.0) (product "bike") (rnd 1.0))
(B::bid (agent "buyer3fe56e01") (amount 112.0) (product "bike") (rnd 1.0))
(B::bid (agent "buyer3fdc06c0") (amount 110.0) (product "bike") (rnd 1.0))
(B::bid (agent "buyer3fe56e01") (amount 142.0) (product "bike") (rnd 2.0))
(B::bid (agent "buyer3fdc06c0") (amount 140.0) (product "bike") (rnd 2.0))
(B::bid (agent "buyer3fe6ecba") (amount 160.0) (product "bike") (rnd 3.0))
(B::winner (agent "buyer3fe6ecba") (amount 160.0))
(B::rnd 0.0)
(B::blacklist "jake")
```

```
(I::rea "nick" "buyer" "buyer3fe56e01")
(I::rea "nick" "buyer")
(I::rea "john" "buyer" "buyer3fdc06c0")
(I::rea "john" "buyer")
(I::rea "jake" "buyer" "buyer3fe6ecba")
(I::rea "jake" "buyer")
(I::rea "marie" "auctioneer" "auctioneer3fd1157c")
(I::rea "marie" "auctioneer")
(I::viol pay1 "buyer3fe6ecba" 160.0)
```

```
(DO::reset "auctioneer3fd1157c")
```

Figure A.5: Using the artifact inspector for inspecting the artifact's internal state.

Bibliography

Aştefănoaei, L., Dastani, M., Meyer, J.-J., and Boer, F. S. (2008). A verification framework for normative multi-agent systems. In *PRIMA 2008*, pages 54–65, Berlin, Heidelberg. Springer-Verlag.

Aştefănoaei, L., Boer, F. S. d., and Dastani, M. (2009a). Rewriting agent societies strategically. In *WI-IAT '09: Proceedings of the 2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology*, pages 441–444, Washington, DC, USA. IEEE Computer Society.

Aştefănoaei, L., Boer, F. S. d., and Dastani, M. (2010). Strategic executions of choreographed timed normative multi-agent systems. In *AAMAS '10: Proceedings of The 9th International Conference on Autonomous Agents and Multiagent Systems*, pages 965–972. IFAAMAS.

Aştefănoaei, L., de Boer, F. S., and Dastani, M. (2009b). On coordination, autonomy and time. In *AAMAS '09: Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems*, pages 1357–1358. IFAAMAS.

Adal, A. (2010). An interpreter for organization oriented programming language (2opl). Master's thesis, Utrecht University.

Alchourron, C. E., Gardenfors, P., and Makinson, D. (1985). On the logic of theory change: Partial meet contraction and revision functions. *Symbolic Logic*, 50:510–530.

Aldewereld, H. (2007). *Autonomy vs. Conformity - an Institutional Perspective on Norms and Protocols.* PhD thesis, SIKS.

Arbab, F., Aştefănoaei, L., Boer, F. S., Dastani, M., Meyer, J.-J., and Tinnemeier, N. (2008). Reo connectors as coordination artifacts in 2apl systems. In *PRIMA '08: Proceedings of the 11th Pacific Rim International Conference on Multi-Agents*, pages 42–53, Berlin, Heidelberg. Springer-Verlag.

Artikis, A. (2003). *Executable Specification of Open Norm-Governed Computational Systems.* PhD thesis, Imperial College London.

Artikis, A. (2009). Dynamic protocols for open agent systems. In *AAMAS*, pages 97–104.

Artikis, A. and Pitt, J. (2001). A formal model of open agent societies. In *Proceedings of the fifth international conference on Autonomous agents*, pages 192–193, New York, NY, USA. ACM.

Artikis, A. and Sergot, M. (2010). Executable specification of open multi-agent systems. *Logic Journal of IGPL*, Advance Access published December 14, 2009.

Artikis, A., Sergot, M., and Pitt, J. (2009). Specifying norm-governed computational societies. *ACM Trans. Comput. Logic*, 10(1):1–42.

Bachman, C. W. and Daya, M. (1977). The role concept in data models. In *VLDB '1977: Proceedings of the third international conference on Very large data bases*, pages 464–476. VLDB Endowment.

Baier, C. and Katoen, J.-P. (2008). *Principles of Model Checking (Representation and Mind Series).* The MIT Press.

Baldoni, M., Boella, G., Genovese, V., Grenna, R., Mugnaini, A., and van der Torre, L. (2009). A middleware for modeling organizations and roles in jade. In *Programming Multi-Agent Systems, First International Workshop, PROMAS 2009,*. Springer-Verlag.

Baldoni, M., Boella, G., and van der Torre, L. (2005). Roles as a coordination construct: Introducing powerjava. In *Proceedings of 1st International Workshop on Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems.*

Baldoni, M., Boella, G., and van der Torre, L. (2007). Interaction between objects in powerjava. *Journal of Object Technology*, 6(2).

Baldoni, M., Genovese, V., Grenna, R., and van der Torre, L. (2008). Adding organizations and roles as primitives to the jade framework. In *Proceedings of the 3rd International Workshop on Normative Multi-Agent Systems.*

Bäumer, D., Riehle, D., Siberski, W., and Wulf, M. (1998). The role object pattern. In *Proceedings of PLoP '97. Technical Report WUCS-97-34. Washington University, Dept. of Computer Science.*

Bellifemine, F. L., Caire, G., and Greenwood, D. (2007). *Developing Multi-Agent Systems with JADE.* Wiley.

Biddle, B. (1986). Recent developments in role theory. *Annual Review of Sociology*, 12(1):67–92.

Boella, G., Pigozzi, G., and van der Torre, L. (2009). Normative systems in computer science - ten guidelines for normative multiagent systems. In Boella, G., Noriega, P., Pigozzi, G., and Verhagen, H., editors, *Normative Multi-Agent Systems*, number 09121 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

Boella, G., Torre, L., and Verhagen, H. (2006). Introduction to normative multiagent systems. *Comput. Math. Organ. Theory*, 12(2-3):71–79.

Boella, G., Torre, L., and Verhagen, H. (2008). Introduction to the special issue on normative multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 17(1):1–10.

Boella, G. and Torre, L. V. D. (2004). Organizations as socially constructed agents in the agent oriented paradigm. In *Proceedings of the Fifth International Workshop on Engineering Societies in the Agents' World (ESAW'04)*, pages 1–13. Springer Verlag.

Boella, G. and van der Torre, L. (2004). Regulative and constitutive norms in normative multiagent systems. In *Proceedings of the Ninth International Conference on the Principles of Knowledge Representation and Reasoning*, pages 255–265. AAAI Press.

Bordini, R. H., Hübner, J. F., and Vieira, R. (2005). Jason and the golden fleece of agent-oriented programming. In Bordini, R. H., Dastani, M., Dix, J., and Fallah-Seghrouchni, A. E., editors, *Multi-Agent Programming*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 3–37. Springer.

Bou, E., López-Sánchez, M., and Rodríguez-Aguilar, J. A. (2006). Adaptation of autonomic electronic institutions through norms and institutional agents. In *Engineering Societies in the Agents' World*, pages 300–319.

Bratman, M. (1987). *Intentions, Plans, and Practical Reason.* Harvard University Press, Cambridge, MA, USA.

Bratman, M., Israel, D., and Pollack, M. (1988). Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4(4):349–355.

Broersen, J., Dastani, M., Hulstijn, J., and Torre, L. v. d. (2002). Goal generation in the BOID architecture. *Cognitive Science Quarterly Journal*, 2(3-4):428–447.

Broersen, J. and van der Torre, L. (2007). Reasoning about norms, obligations, time and agents. In *PRIMA*, pages 171–182.

Campos, J., López-Sánchez, M., Rodríguez-Aguilar, J. A., and Esteva, M. (2009). Formalising situatedness and adaptation in electronic institutions. In *Coordination, Organizations, Institutions and Norms in Agent Systems V*, pages 126–139, Berlin, Germany. Springer.

Cap, M. (2010). Belief/goal sharing bdi modules. Master's thesis, Utrecht University.

Castelfranchi, C. (2000). Engineering social order. In *Proceedings of the First International Workshop on Engineering Societies in the Agents' World (ESAW'00)*, pages 1–18, London, UK. Springer.

Castelfranchi, C. (2003). The micro-macro constitution of power. *Journal of Protosociology*, 1(18):208–269.

Castelfranchi, C. (2004). Formalizing the informal?: Dynamic social order, bottom-up social control, and spontaneous normative relations. *Journal of Applied Logic*, 1(1-2):47–92.

Chellas, B. F. (1980). *Modal Logic: An Introduction.* Cambridge University Press, Cambridge, USA.

Cohen, P. R. and Levesque, H. J. (1990). Intention is choice with commitment. *Artificial Intelligence*, 42(2-3):213–261.

Colman, A. and Han, J. (2007). Roles, players and adaptable organizations. *Applied Ontology*, 2(2):105–126.

Coutinho, L., Sichman, J. S., and Boissier, O. (2005). Modeling organization in mas: a comparison of models. In *Proceedings of the 1st. Workshop on Software Engineering for Agent-Oriented Systems.*

Cremonini, M., Omicini, A., and Zambonelli, F. (2000). Coordination and access control in open distributed agent systems: The tucson approach. In *COORDINATION '00: Proceedings of the 4th International Conference on Coordination Languages and Models*, pages 99–114, London, UK. Springer-Verlag.

d'Altan, P., Meyer, J.-J. C., and Wieringa, R. (1996). An integrated framework for ought-to-be and ought-to-do constraints. *AI & Law*, 4(2):77–111.

Dastani, M. (2008). 2APL: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248.

Dastani, M., Dignum, V., and Dignum, F. (2003). Role-assignment in open agent societies. In *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 489–496, New York, NY, USA. ACM.

Dastani, M., Dix, J., and Fallah-Seghrouchni, A. E., editors (2004a). *Programming Multi-Agent Systems, First International Workshop, PROMAS 2003, Melbourne, Selected Revised and Invited Papers*, volume 3067 of *Lecture Notes in Computer Science.* Springer.

Dastani, M., Grossi, D., Meyer, J.-J. C., and Tinnemeier, N. A. M. (2008). Normative multi-agent programs and their logics. In *KRAMAS*, pages 16–31.

Dastani, M. and Steunebrink, B. (2009). Operational semantics for bdi modules in multi-agent programming. In *Proceedings 10th Workshop on Computational Logic in Multi-Agent Systems (CLIMA-X)*.

Dastani, M., Tinnemeier, N. A. M., and Meyer, J.-J. C. (2009). *A Programming Language for Normative Multi-Agent Systems*, chapter 16. In Dignum (2009a).

Dastani, M., van Riemsdijk, M., Hulstijn, J., F.Dignum, and Meyer, J.-J. C. (2004b). Enacting and deacting roles in agent programming. In *Proceedings of the 5th International Workshop on Agent-Oriented Software Engineering*, page 3382.

Davidsson, P. (2001). Categories of artificial societies. In *Proceedings of the Second International Workshop on Engineering Societies in the Agents World II*, pages 1–9, London, UK. Springer-Verlag.

de Boer, F., Hindriks, K., van der Hoek, W., and Meyer, J.-J. (2007). A Verification Framework for Agent Programming with Declarative Goals. *Journal of Applied Logic*.

Dennett, D. (1987). *The Intentional Stance*. The MIT Press, Cambridge, USA.

Dennis, L., N.A.M., T., and Ch., M. J.-J. (2009). Model checking normative agent organisations. In *Proceedings 10th Workshop on Computational Logic in Multi-Agent Systems (CLIMA-X)*.

Dignum, F. (1999). Autonomous agents with norms. *AI & Law*, 7:69–79.

Dignum, F., Broersen, J., Dignum, V., and Meyer, J.-J. (2004). Meeting the deadline: Why, when and how. In *Proceedings of the Third Conference on Formal Aspects of Agent-Based Systems*, pages 30–40. Springer Verlag.

Dignum, V. (2004). *A Model for Organizational Interaction: Based on Agents, Founded in Logic*. PhD thesis, Utrecht University, SIKS.

Dignum, V., editor (2009a). *Handbook of Research on Multi-Agent Systems: Semantics and Dynamics of Organizational Models*. IGI Global.

Dignum, V. (2009b). *The Role of Organization in Agent Systems*, chapter 1, pages 1–16. IGI Global.

Eker, S., Martí-Oliet, N., Meseguer, J., and Verdejo, A. (2007). Deduction, strategies, and rewriting. *Electron. Notes Theor. Comput. Sci.*, 174(11):3–25.

Elhag, A. A. O., Breuker, J. A. P. J., and Brouwer, P. W. (2000). On the formal analysis of normative conflicts. *Information and Communication Technology Law*, 9(3):207–217.

Emerson, E. (1990). Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 995–1072. MIT Press, Cambridge, MA, USA.

Esteva, M., de la Cruz, D., and Sierra, C. (2002). Islander: an electronic institutions editor. In *AAMAS'02: Proceedings of The 1st International Conference on Autonomous Agents and Multiagent Systems*, pages 1045–1052.

Esteva, M., Rodríguez-Aguilar, J., Rosell, B., and Arcos, J. (2004). Ameli: An agent-based middleware for electronic institutions. In *AAMAS'04: Proceedings of The 3th International Conference on Autonomous Agents and Multiagent Systems*, New York, US.

Esteva, M., Rodríguez-Aguilar, J., Sierra, C., Garcia, P., and Arcos, J. (2001). On the formal specifications of electronic institutions. In *Agent Mediated Electronic Commerce, The European AgentLink Perspective.*, pages 126–147, London, UK. Springer.

Fenton, N. and Pfleeger, S. (1998). *Software Metrics: A Rigorous and Practical Approach, Revised.* Course Technology.

Ferber, J., Gutknecht, O., and Michel, F. (2003). From agents to organizations: An organizational view of multi-agent systems. In *AOSE'03: 4th International Workshop on Agent-Oriented Software Engineering*, pages 214–230.

Fornara, N. and Colombetti, M. (2009). *Specifying Artificial Institutions in the Event Calculus*, chapter 14. In Dignum (2009a).

Gaertner, D., Garcia-Camino, A., Noriega, P., Rodriguez-Aguilar, J.-A., and Vasconcelos, W. (2007). Distributed norm management in regulated multiagent systems. In *AAMAS '07: Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, pages 1–8, New York, NY, USA. ACM.

Garcia-Camino, A., Noriega, P., and Rodriguez-Aguilar, J. A. (2005). Implementing norms in electronic institutions. In *AAMAS '05: Proceedings of The 4th International Conference on Autonomous Agents and Multiagent Systems*, pages 667–673, New York, NY, USA. ACM.

Georgeff, M., Pell, B., Pollack, M., Tambe, M., and Wooldridge, M. (1999). The Belief-Desire-Intention Model of Agency. In *ATAL '98: Proceedings of the 5th International Workshop on Intelligent Agents, Agent Theories, Architectures, and Languages*, pages 1–10, London, UK. Springer-Verlag.

Georgeff, M. P. and Lansky, A. L. (1987). Reactive reasoning and planning. In *AAAI*, pages 677–682.

Governatori, G. and Rotolo, A. (2008). Changing legal systems: Abrogation and annulment. Part I: Revision of defeasible theories. In *DEON*, pages 3–18. Springer.

Grossi, D. (2007). *Designing Invisible Handcuffs. Formal Investigations in Institutions and Organizations for MAS*. PhD thesis, Utrecht University, SIKS.

Grossi, D., Dignum, F., Dastani, M., and Royakkers, L. (2005). Foundations of organizational structures in multiagent systems. In *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 690–697, New York, NY, USA. ACM.

Hansen, J., Pigozzi, G., and van der Torre, L. (2007). Ten philosophical problems in deontic logic. In *NorMAS*, Dagstuhl, Germany. IBFI.

Hewitt, C. (1986). Offices are open systems. *ACM Trans. Inf. Syst.*, 4(3):271–287.

Hiel, M., Aldewereld, H., and Frank, D. (2010). Modeling warehouse logistics using agent organizations. In *CARE 2010: Second International Workshop on Collaborative Agents – REsearch and Development*.

Hindriks, K. V., De Boer, F. S., Van Der Hoek, W., and Ch. Meyer, J.-J. (1999). Agent programming in 3apl. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401.

Hübner, J. F., Boissier, O., and Bordini, R. H. (2010). A normative organisation programming language for organisation management infrastructures. In *Coordination, Organizations, Institutions and Norms in Agent Systems V*, pages 114–129. Springer.

Hübner, J. F., Bordini, R. H., and Wooldridge, M. (2006a). Plan patterns for declarative goals in agentspeak. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1291–1293, New York, NY, USA. ACM.

Hübner, J. F., Sichman, J. S., and Boissier, O. (2007). Developing organised multiagent systems using the moise+ model: programming issues at the system and agent levels. *Agent-Oriented Software Engineering*, 1(3/4):370–395.

Hübner, J. F., Sichman, J. S. a., and Boissier, O. (2002). Moise+: towards a structural, functional, and deontic model for mas organization. In *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 501–502, New York, NY, USA. ACM.

Hübner, J. F., Sichman, J. S. a., and Boissier, O. (2006b). S-moise+: A middleware for developing organised multi-agent systems. In *Coordination, Organizations, Institutions and Norms in Agent Systems I*, pages 64–78. Springer.

Jennings, N. R. (2000). On agent-based software engineering. *AI*, 117(2):277–296.

Jones, A. and Sergot, M. (1996). A formal characterization of institutional power. *Logic Journal of the IGPL*, 4(3):429–445.

Jones, A. J. I. and Sergot, M. (1993). On the characterisation of law and computer systems: The normative systems perspective. In *Deontic Logic in Computer Science: Normative System Specification*, pages 275–307. John Wiley & Sons.

Kendall, E. A. (1999). Aspect-oriented programming for role models. In *Proceedings of the Workshop on Object-Oriented Technology*, pages 294–295, London, UK. Springer-Verlag.

Kristensen, B. (1995). Object-oriented modeling with roles. In *Proceedings of the 2nd International Conference on Object-Oriented Information Systems*, pages 57–71. Springer-Verlag.

Lopez y Lopez, F., Luck, M., and d'Inverno, M. (2006). A normative framework for agent-based systems. *Comp. & Math. Org. Theory*, 12(2-3):227–250.

Luck, M., McBurney, P., and Preist, C. (2003). *Agent Technology: Enabling next generation computing: a roadmap for agent based computing.* Agentlink.

McArthur, R. P. (1981). Anderson's Deontic Logic and Relevant Implication. *Notre Dame Journal of Formal Logic*, 22(2):145–154.

Meneguzzi, F. and Luck, M. (2009). Norm-based behaviour modification in BDI agents. In *AAMAS'09: Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems.* IFAAMAS.

Meyer, J.-J. C. (1987). A different approach to deontic logic: deontic logic viewed as a variant of dynamic logic. *Notre Dame J. Formal Logic*, 29(1):109–136.

Meyer, J.-J. C., Dignum, F., and Wieringa, R. (1994). The paradoxes of deontic logic revisited: A computer science perspective (or: Should computer scientists be bothered by the concerns of philosophers?). Technical Report UU-CS-1994-38, Department of Information and Computing Sciences, Utrecht University.

Meyer, J.-J. C. and Wieringa, R. J., editors (1993). *Deontic logic in computer science: normative system specification.* John Wiley & Sons, Inc., New York, NY, USA.

Minsky, N. H. and Ungureanu, V. (2000). Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *ACM Trans. Softw. Eng. Methodol.*, 9(3).

Moses, Y. and Tennenholtz, M. (1995). Artificial social systems. *Computers and AI*, 14:533–562.

Odell, J. J., Van, H., Parunak, D., and Fleischer, M. (2003). The role of roles in designing effective agent organizations. In *Software Engineering for Large-Scale Multi-Agent Systems*, pages 27–38. Springer.

Omicini, A., Ricci, A., Viroli, M., Castelfranchi, C., and Tummolini, L. (2004). Coordination artifacts: Environment-based coordination for intelligent agents. In *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 286–293, Washington, DC, USA. IEEE Computer Society.

Omicini, A. and Zambonelli, F. (1999). Tuple centres for the coordination of internet agents. In *SAC '99: Proceedings of the 1999 ACM symposium on Applied computing*, pages 183–190, New York, NY, USA. ACM.

Oren, N., Luck, M., and Miles, S. (2010). A model of normative power. In *AAMAS '10: Proceedings of The 9th International Conference on Autonomous Agents and Multiagent Systems*, pages 815–822. IFAAMAS.

Park, J. and Sandhu, R. (2002). Towards usage control models: beyond traditional access control. In *Proceedings of the seventh ACM symposium on Access control models and technologies*, pages 57–64, New York, NY, USA. ACM.

Plotkin, G. (1981). A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus.

Pokahr, A., Braubach, L., and Lamersdorf, W. (2005). Jadex: A bdi reasoning engine. In Bordini, R. H., Dastani, M., Dix, J., and Fallah-Seghrouchni, A. E., editors, *Multi-Agent Programming*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 149–174. Springer.

Prakken, H. and Sergot, M. J. (1996). Contrary-to-duty obligations. *Studia Logica*, 57(1):91–115.

Quillinan, T., Brazier, F., Aldewereld, H., Dignum, V., Dignum, F., Penserini, L., and Wijngaards, N. (2009). Developing agent-based organizational models for crisis management. In *AAMAS '09: Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems*.

Rao, A. S. (1996). AgentSpeak(L): BDI agents speak out in a logical computable language. In van Hoe, R., editor, *MAAMAW 1996: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, Eindhoven, The Netherlands.

Rao, A. S. and Georgeff, M. P. (1991). Modelling rational agents within a BDI architecture. In *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning*, pages 473–484, Cambridge, MA, USA.

Ricci, A., Viroli, M., and Omicini, A. (2006). Programming MAS with artifacts. In Bordini, R. P., Dastani, M., Dix, J., and El Fallah Seghrouchni, A., editors, *Programming Multi-Agent Systems, Third International Workshop, PROMAS 2005, Utrecht, The Netherlands. Revised and Invited Papers*, LNAI, pages 206–221. Springer.

Ricci, A., Viroli, M., and Omicini, A. (2007). "Give agents their artifacts": The A&A approach for engineering working environments in MAS. In Durfee, E., Yokoo, M., Huhns, M., and Shehory, O., editors, *AAMAS '07: Proceedings of The 6th International Conference on Autonomous Agents and Multiagent Systems*, pages 601–603. IFAAMAS.

Sammet, J. E. (1971). Problems in, and a pragmatic approach to, programming language measurement. In *Proceedings of the November fall joint computer conference*, pages 243–251, New York, NY, USA. ACM.

Sandhu, R., Coyne, E., Feinstein, H., and Youman, C. (1996). Role-based access control models. *Computer*, 29(2):38–47.

Searle, J. R. (1995). *The Construction of Social Reality.* Free Press.

Sergot, M. J. and Craven, R. (2006). The deontic component of action language nc+. In *DEON*, pages 222–237.

Shoham, Y. (1993). Agent-Oriented Programming. *AI*, 60(1):51–92.

Shoham, Y. and Tennenholtz, M. (1995). On social laws for artificial agent societies: off-line design. *AI*, 73(1-2):231–252.

Silva, V. (2008). From the specification to the implementation of norms: an automatic approach to generate rules from norms to govern the behavior of agents. *JAAMAS*, 17(1):113–155.

Singh, M. P. (1999). An ontology for commitments in multiagent systems: Toward a unification of normative concepts. *Artificial Intelligence and Law*, 7:97–113.

Steimann, F. (2000). On the representation of roles in object-oriented and conceptual modelling. *Data Knowledge Engineering*, 35(1):83–106.

Tambe, M. and Zhang, W. (1998). Towards flexible teamwork in persistent teams. In *ICMAS '98: Proceedings of the 3rd International Conference on Multi Agent Systems*, pages 277–284, Washington, DC, USA. IEEE Computer Society.

Tinnemeier, N. A. M., Dastani, M., and Meyer, J.-J. C. (2009a). Orwell's nightmare for agents? programming multi-agent organisations. In *Programming Multi-Agent Systems: 6th International Workshop. Revised, Invited and Selected Papers*, pages 56–71, Berlin, Heidelberg. Springer-Verlag.

Tinnemeier, N. A. M., Dastani, M., and Meyer, J.-J. C. (2009b). Roles and norms for programming agent organizations. In *AAMAS '09: Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems*, pages 121–128, Richland, SC. International Foundation for Autonomous Agents and Multiagent Systems.

Tinnemeier, N. A. M., Dastani, M., and Meyer, J.-J. C. (2010). Programming norm change. In *AAMAS '10: Proceedings of The 9th International Conference on Autonomous Agents and Multiagent Systems*, pages 957–964. IFAAMAS.

Tinnemeier, N. A. M., Dastani, M. M., Meyer, J.-J. C., and van der Torre, L. (2009c). Programming normative artifacts with declarative obligations and prohibitions. In *Web Intelligence and Intelligent Agent Technology, IEEE/WIC/ACM International Conference on*, pages 145–152, Los Alamitos, CA, USA. IEEE Computer Society.

Torroni, P., Yolum, P., Singh, M. P., Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., and Mello, P. (2009). *Modelling Interactions via Commitments and Expectations*, chapter 11, pages 263–284. In Dignum (2009a).

Van der Vecht, B. (2009). *Adjustable Autonomy: Controling Influences on Decision Making.* PhD thesis, Utrecht University, SIKS.

van Riemsdijk, M. B., Dastani, M., and Meyer, J.-J. C. (2005). Semantics of declarative goals in agent programming. In *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 133–140, New York, NY, USA. ACM.

van Riemsdijk, M. B., Dastani, M., and Winikoff, M. (2008). Goals in agent systems: a unifying framework. In *AAMAS '08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, pages 713–720, Richland, SC. International Foundation for Autonomous Agents and Multiagent Systems.

van Riemsdijk, M. B., Hindriks, K. V., and Jonker, C. M. (2009). Programming organization-aware agents: A research agenda. In *Proceedings of the Tenth International Workshop on Engineering Societies in the Agents' World (ESAW'09)*, volume 5881 of *LNAI*, pages 98–112. Springer.

van Riemsdijk, M. B., Hindriks, K. V., Jonker, C. M., and Sierhuis, M. (2010). Formalizing organizational constraints: A semantic approach. In *AAMAS '10: Proceedings of the ninth international joint conference on Autonomous agents and multiagent systems*, pages 823–830. ACM.

Vasconcelos, W., Kollingbaum, M. J., and Norman, T. J. (2007). Resolving conflict and inconsistency in norm-regulated virtual organizations. In *AAMAS '07: Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, pages 1–8, New York, NY, USA. ACM.

Vasconcelos, W. W., Kollingbaum, M. J., and Norman, T. J. (2009). Normative conflict resolution in multi-agent systems. *JAAMAS*, 19(2):124–152.

Vázquez-Salceda, J. (2004). *The role of norms and electronic institutions in multi-agent systems. The HARMONIA framework.* Whitestein Series in Software Agent Technologies. Birkhäuser Verlag, Basel, Switzerland.

Vázquez-Salceda, J., Aldewereld, H., and Dignum, F. (2004). Implementing norms in multiagent systems. In *Proceedings of the Second German Conference on Multiagent System Technologies*, pages 313–327.

Vázquez-Salceda, J., Aldewereld, H., Grossi, D., and Dignum, F. (2008). From human regulations to regulated software agents' behavior. *AI & Law*, 16(1):73–87.

von Wright, G. H. (1951). Deontic logic. *Mind*, 60(237):1–15.

Watt, D. A. (2004). *Programming Language Design Concepts.* John Wiley & Sons.

Weyns, D., Omicini, A., and Odell, J. (2007). Environment as a first class abstraction in multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 14(1):5–30.

Wing, J. M. (2008). Five deep questions in computing. *Communications of the ACM*, 51(1):58–60.

Winikoff, M. (2009). Future directions for agent-based software engineering. *Agent-Oriented Software Engineering*, 3(4):402–410.

Wooldridge, M. and Jennings, N. (1995). Intelligent Agents: Theory and Practice. *The Knowledge Engineering Review*, 10(2):115–152.

Wooldridge, M., Jennings, N. R., and Kinny, D. (2000). The gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312.

Wooldridge, M. J. (2001). *Introduction to Multiagent Systems.* John Wiley & Sons, Inc., New York, NY, USA.

Yolum, P. and Singh, M. P. (2002). Flexible protocol specification and execution: applying event calculus planning using commitments. In *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 527–534, New York, NY, USA. ACM.

Zambonelli, F., Jennings, N. R., and Wooldridge, M. (2001a). Organisational rules as an abstraction for the analysis and design of multi-agent systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(3):303–328.

Zambonelli, F., Jennings, N. R., and Wooldridge, M. (2001b). Organizational abstractions for the analysis and design of multi-agent system. In *AOSE'00: First international workshop Agent-oriented software engineering*, pages 235–251, New York, NJ, USA. Springer-Verlag.

Zambonelli, F., Jennings, N. R., and Wooldridge, M. (2003). Developing multiagent systems: The gaia methodology. *ACM Trans. Softw. Eng. Methodol.*, 12(3):317–370.

Summary

The history of software engineering in general and programming languages in particular is marked by the introduction of high-level engineering concepts, abstracting away from the rather low-level principles that are used by the machine on which the software is executed. Such high-level abstractions allow us to focus only on a few essential concepts at the same time by factoring out details. The abstractions by which we engineer complex software systems are more than often inspired by metaphorical concepts by which we understand and structure the complex world around us. A well-known example is the concept of folder for archiving our files. Introducing the concepts of agent as the metaphorical counterpart of humans and multi-agent system as the metaphorical counterpart of a society, the field of agent-oriented software engineering brings the use of abstractions in programming to an even higher level.

Agents are autonomous entities that are typically programmed in terms of beliefs modeling the information they have about their world, goals denoting the situations they desire to establish and plans describing how to reach their goals. Agents participating in a multi-agent system may have been engineered by different parties with differing design objectives, implying that agents may encounter and interact with agents having conflicting goals. An illustrative example is an online marketplace on which agents interact with (unknown) parties to sell and buy their goods. Because little can be assumed about the behavior the interacting agents exhibit and nobody directly wrote the whole program encompassing the multi-agent system it is hard to predict the emerging behavior of the system as a whole. To increase the likelihood of the design objectives of the system being met, coordination media are put in place to regulate the individual agents' behavior. Some of these media are based on constructs that resemble physical every-day structures, such as a tube's entrance gate and traffic lights. Yet others use more abstract concepts we use for organizing our society, such as norms that should be followed and roles that agents can play. Getting back to the online marketplace example, agents play the role of seller and buyer, and are expected to abide by certain norms, e.g. paying the price agreed within a certain time. In this thesis we will focus on organization-oriented coordination media.

In this thesis we show that research on individual agents progressed rather independently from research on agent organizations, leaving a gap between agent-oriented and organization-oriented programming. We identify what we consider the root causes underlying this gap and develop an organization-oriented programming language whose constructs accord better with the key concepts and characteristics associated with agents. Constructs for programming roles, norms and constructs for changing the norms at runtime will be investigated in particular. To understand what our programming language can (or cannot) offer, a precise description of its meaning (semantics) is indispensable. For example, to use an obligation properly, we need to know exactly when it is fulfilled or violated, and when sanctions will be imposed. Therefore, in this thesis, we formally describe the semantics of the programming constructs in a mathematically rigorous manner.

Samenvatting

De geschiedenis van software engineering in het algemeen, en die van programmeertalen in het bijzonder, wordt gekenmerkt door het gebruik van abstracte concepten. Deze concepten gaan voorbij aan de principes die door de machine waarop de software wordt uitgevoerd, worden gebruikt. Het gebruik van dergelijke abstracties staat ons toe om ons slechts te hoeven concentreren op de essentiële zaken, zonder ons te hoeven bekommeren om details. De abstracties waarmee wij complexe software bouwen zijn vaak gebaseerd op metaforen van alledaagse concepten, die we ook gebruiken om onze complexe wereld te begrijpen en te structureren. Het concept map (folder) voor het archiveren van bestanden (files) is een bekend voorbeeld hiervan. Met de introductie van het concept agent als metaforische tegenhanger van mens, en multi-agent systeem als metaforische tegenhanger van maatschappij, wordt het gebruik van abstracties op het gebied van agent-georiënteerde softwaretechnologie in software ontwikkeling op een nog hoger niveau gebracht.

Agenten zijn autonome entiteiten die veelal geprogrammeerd worden in termen van kennis, die zij hebben over hun wereld, doelen die de situaties aanduiden die zij wensen te bereiken en plannen die beschrijven hoe ze hun doelen kunnen bereiken. Agenten die onderdeel vormen van een multi-agent systeem zijn mogelijk ontworpen door verschillende ontwikkelaars met verschillende ontwerpdoelen en kunnen dan ook agenten ontmoeten met doelen die conflicteren met hun eigen doelen. Een illustratief voorbeeld is dat van een online marktplaats, waarop de agenten met (onbekende) partijen interacteren om hun goederen te kopen en verkopen. Agenten spelen de rol van koper en verkoper en worden verwacht aan bepaalde normen te voldoen, bijvoorbeeld het tijdig betalen van een gekocht product. Omdat weinig over het gedrag van deze agenten kan worden verondersteld en het gehele multi-agent systeem niet door één persoon geschreven is, is het moeilijk om het uiteindelijke gedrag van het gehele systeem te voorspellen.

Om de kans te vergroten dat de ontwerpdoelen van het systeem worden behaald, worden zogeheten coördinatiemedia ingezet om het gedrag van individuele agenten te regelen. Sommige van deze coördinatiemedia zijn gebaseerd op fysieke coördinatiestructuren uit het dagelijks leven, zoals tolpoortjes en verkeerslichten. Andere zijn gebaseerd op concepten van hogere abstractie die we gebruiken voor het organiseren van onze maatschappij, zoals normen die gevolgd dienen te worden en rollen die gespeeld kunnen worden.

In dit proefschrift richten we ons op organisatie-georiënteerde coördinatiemedia, waarbij we aantonen dat de gebieden van individuele agenten en agent organisaties onafhankelijk van elkaar zijn onderzocht, waardoor er een hiaat is ontstaan tussen het programmeren van agenten en agent organisaties. We identificeren de belangrijkste oorzaken die hier aan ten grondslag liggen en ontwikkelen een organisatie-georiënteerde programmeertaal met constructies die beter aansluiten bij de karakteristieken en concepten van agenten. We zullen in het bijzonder constructen voor het programmeren van rollen, normen en voor het veranderen van de normen onderzoeken. Om te begrijpen wat onze programmeertaal te bieden heeft, is een nauwkeurige beschrijving van zijn betekenis (semantiek) onontbeerlijk. Om een verplichting te gebruiken moet het bijvoorbeeld duidelijk zijn wanneer deze vervuld dan wel geschonden is en wanneer eventuele sancties opgelegd zullen worden. Om deze reden beschrijven we de semantiek van de voorgestelde constructies nauwkeurig in een formeel raamwerk.

Dankwoord

Vier jaar noeste arbeid en 4380 koppen koffie later kan ik nog steeds geen mensen genezen, zal mijn salaris de Balkenende norm niet overschrijden en zal ik niet deelnemen aan het boekenbal. Toch mag ik mij straks doctor noemen, zal ik gepromoveerd zijn en heb ik een boek geschreven. Hoewel alleen mijn naam op de kaft staat had ik dit boek niet kunnen schrijven zonder de hulp van anderen. En daar wil ik hier iedereen warm voor bedanken.

Ik wil graag beginnen met het bedanken van mijn promotor, John-Jules Meyer. Door je diepgaande kennis gecombineerd met je altijd optimistische, open blik waren onze vergaderingen altijd inspirerend en prettig. Ook wist je me telkens weer te verbazen met openbaringen uit je persoonlijke leven. Door je aanstekelijke schaterlach die altijd ver over de IS gang te horen valt, waren ook vergaderingen met anderen een prettige ervaring.

Mehdi Dastani, vanaf de eerste dag heb je me overal bij betrokken: van het geven van colleges tot het meeschrijven aan wetenschappelijke artikelen. Op de vraag die me ooit over je gesteld werd: “is het niet moeilijk om hem bij te houden?”, kon ik alleen met een volmondig “ja!” antwoorden. Van de hoeveelheid uiteenlopend onderzoek die je doet en het rijke sociale leven dat je er daarnaast op na houdt ben ik altijd nog onder de indruk. Ik had me geen betere co-promotor kunnen wensen!

Frank de Boer, je hebt je bij John-Jules en Mehdi gevoegd toen mijn proefschrift eigenlijk al geschreven was. Dat is niet raar, want we hebben aan hetzelfde project gewerkt en er heeft veel kruisbestuiving plaatsgevonden tussen ons werk en dat van jullie. Bij deze wil ik je bedanken voor het lezen en beoordelen van mijn proefschrift en ik ben vereerd dat ik je als co-promotor mag noemen!

Ook mijn afstudeerbegeleiders, Jan Kuper en Philip Hölzenspies, mogen zeker niet aan deze lijst ontbreken. Zonder jullie had ik nooit aan deze vier jaren onderzoek kunnen beginnen. En hoewel ik er toen echt helemaal anders over

dacht, moet ik nu toch toegeven dat het compleet herschrijven van mijn scriptie niet alleen wreed, maar eigenlijk ook wel nuttig was. Mede hierdoor is mijn proefschrift nu mooi op tijd af.

Mijn leescommissie wil ik bedanken voor het lezen en (positief) beoordelen van mijn proefschrift. Met name Leon van der Torre ben ik veel dank verschuldigd. Je hebt mijn proefschrift nauwgezet gelezen en je zeer uitgebreide inzichten en suggesties hebben mijn proefschrift zonder twijfel verbeterd. Ook heb ik erg genoten van ons gezamenlijk onderzoek en de inzichtelijke discussies onder het genot van een lekker glaasje wijn en een stukje kaas.

Louise Dennis en Michael Fisher bedank ik voor de uitnodiging en inspirerende samenwerking tijdens ons gezamenlijk onderzoek binnen de Logic and Computation groep van de universiteit van Liverpool.

Frank Dignum, Virginia Dignum, Huib Aldewereld, ik heb onze open discussies over mijn (en jullie) werk altijd zeer gewaardeerd. Het is goed om ook andere invalshoeken te horen en te realiseren dat er niet één waarheid is. Het is jammer dat onze samenwerking beperkt is gebleven tot discussies.

Collega's en ex-collega's, kleine kans dat ik niet een van mijn 4380 koppen koffie met jullie genoten heb. Met jullie heb ik niet alleen aan mijn academische vaardigheden gewerkt, maar heb ik vooral nieuwe talenten kunnen ontwikkelen: van tafeltennissen tot het in bomen hangen in de botanische tuinen, van het (vals) zingen in een virtuele rockband tot het amoveren van een Brits college.

Lieve vrienden, zusje, zwager en schoonmoeder. Bedankt dat jullie al die jaren belasting betaald hebben om mij deze linkse hobby te kunnen laten beoefenen! Maar vooral bedankt dat jullie altijd op de juiste momenten (even niet) vroegen hoe het er met mijn onderzoek voorstond en ik altijd aan kon kloppen voor een openhartig advies.

Lieve opa en oma, ook jullie wil ik voor jullie steun en interesse bedanken. Ook al moet het voor jullie soms geleken hebben of ik vier jaar lang alleen maar verre reizen heb gemaakt. Oma, ik zal de sms(!) die ik van je ontving in Toronto nooit vergeten. Opa, ik weet dat je graag bij mijn verdediging aanwezig had willen zijn.

Pap, mam, ook jullie ben ik veel dank verschuldigd. Jullie hebben mij op liefdevolle wijze gestimuleerd om mijn kwaliteiten ten volle te benutten, zonder daarbij uit het oog te verliezen vooral ook te genieten. Ik weet zeker dat we samen van jullie eerste wereldreis gaan genieten!

Nu, vier jaren en 4380 koppen koffie later klim ik omhoog naar de bovenste plank van de boekenkast. Trots zet ik mijn proefschrift erop en klim snel weer naar beneden. Mijn werk is gedaan, en daar drink ik mijn 4381ste koffie op! En die drink ik niet alleen, die drink ik met mijn allerliefste. Marlies, zonder jou zouden de afgelopen vier jaren nooit zo leuk zijn geweest! (En mijn proefschrift nu waarschijnlijk ook nog lang niet af.)

Curriculum Vitae

Nick Antonius Marinus Tinnemeier

07-11-1980	Born in Enschede, The Netherlands
1993-1999	High School (Atheneum) *Rietveld Lyceum*, Doetinchem, The Netherlands
1999-2003	Bachelor (with honor) Information and Communication Technology *Hogeschool van Arnhem en Nijmegen*, Arnhem, The Netherlands
2003-2006	Master of Science (with honor) Computer Science *University of Twente*, Enschede, The Netherlands
2006-2010	Doctor of Philosophy Agent Technology *Utrecht University*, Utrecht, The Netherlands

SIKS Dissertation Series

1998

1998-1, **Johan van den Akker** (CWI), *DEGAS - An Active, Temporal Database of Autonomous Objects.*

1998-2, **Floris Wiesman** (UM), *Information Retrieval by Graphically Browsing Meta-Information.*

1998-3, **Ans Steuten** (TUD), *A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective.*

1998-4, **Dennis Breuker** (UM), *Memory versus Search in Games.*

1998-5, **E.W.Oskamp** (RUL), *Computerondersteuning bij Straftoemeting.*

1999

1999-1, **Mark Sloof** (VU), *Physiology of Quality Change Modelling; Automated modelling of Quality Change of Agricultural Products.*

1999-2, **Rob Potharst** (EUR), *Classification using decision trees and neural nets.*

1999-3, **Don Beal** (UM), *The Nature of Minimax Search.*

1999-4, **Jacques Penders** (UM), *The practical Art of Moving Physical Objects.*

1999-5, **Aldo de Moor** (KUB), *Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems.*

1999-6, **Niek J.E. Wijngaards** (VU), *Re-design of compositional systems.*

1999-7, **David Spelt** (UT), *Verification support for object database design.*

1999-8, **Jacques H.J. Lenting** (UM), *Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation..*

2000

2000-1, **Frank Niessink** (VU), *Perspectives on Improving Software Maintenance.*

2000-2, **Koen Holtman** (TUE), *Prototyping of CMS Storage Management.*

2000-3, **Carolien M.T. Metselaar** (UVA), *Sociaal-organisatorische gevolgen van kennistechnologie; een procesbenadering en actorperspectief..*

2000-4, **Geert de Haan** (VU), *ETAG, A Formal Model of Competence Knowledge for User Interface Design.*

2000-5, **Ruud van der Pol** (UM), *Knowledge-based Query Formulation in Information Retrieval..*

2000-6, **Rogier van Eijk** (UU), *Programming Languages for Agent Communication.*

2000-7, **Niels Peek** (UU), *Decision-theoretic Planning of Clinical Patient Management.*

2000-8, **Veerle Coup** (EUR), *Sensitivity Analyis of Decision-Theoretic Networks.*

2000-9, **Florian Waas** (CWI), *Principles of Probabilistic Query Optimization.*

2000-10, **Niels Nes** (CWI), *Image Database Management System Design Considerations, Algorithms and Architecture.*

2000-11, **Jonas Karlsson** (CWI), *Scalable Distributed Data Structures for Database Management.*

2001

2001-1, **Silja Renooij** (UU), *Qualitative Approaches to Quantifying Probabilistic Networks.*

2001-2, **Koen Hindriks** (UU), *Agent Programming Languages: Programming with Mental Models.*

2001-3, **Maarten van Someren** (UvA), *Learning as problem solving.*

2001-4, **Evgueni Smirnov** (UM), *Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets.*

2001-5, **Jacco van Ossenbruggen** (VU), *Processing Structured Hypermedia: A Matter of Style.*

2001-6, **Martijn van Welie** (VU), *Task-based User Interface Design.*

2001-7, **Bastiaan Schonhage** (VU), *Diva: Architectural Perspectives on Information Visualization.*

2001-8, **Pascal van Eck** (VU), *A Compositional Semantic Structure for Multi-Agent Systems Dynamics..*

2001-9, **Pieter Jan 't Hoen** (RUL), *Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes.*

2001-10, **Maarten Sierhuis** (UvA), *Modeling and Simulating Work Practice BRAHMS: a multiagent modeling and simulation language for work practice analysis and design.*

2001-11, **Tom M. van Engers** (VUA), *Knowledge Management: The Role of Mental Models in Business Systems Design.*

2002

2002-01, **Nico Lassing** (VU), *Architecture-Level Modifiability Analysis.*

2002-02, **Roelof van Zwol** (UT), *Modelling and searching web-based document collections.*

2002-03, **Henk Ernst Blok** (UT), *Database Optimization Aspects for Information Retrieval.*

2002-04, **Juan Roberto Castelo Valdueza** (UU), *The Discrete Acyclic Digraph Markov Model in Data Mining.*

2002-05, **Radu Serban** (VU), *The Private Cyberspace Modeling Electronic Environments inhabited by Privacy-concerned Agents.*

2002-06, **Laurens Mommers** (UL), *Applied legal epistemology; Building a knowledge-based ontology of the legal domain.*

2002-07, **Peter Boncz** (CWI), *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications.*

2002-08, **Jaap Gordijn** (VU), *Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas.*

2002-09, **Willem-Jan van den Heuvel** (KUB), *Integrating Modern Business Applications with Objectified Legacy Systems.*

2002-10, **Brian Sheppard** (UM), *Towards Perfect Play of Scrabble.*

2002-11, **Wouter C.A. Wijngaards** (VU), *Agent Based Modelling of Dynamics: Biological and Organisational Applications.*

2002-12, **Albrecht Schmidt** (Uva), *Processing XML in Database Systems.*

2002-13, **Hongjing Wu** (TUE), *A Reference Architecture for Adaptive Hypermedia Applications.*

2002-14, **Wieke de Vries** (UU), *Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems.*

2002-15, **Rik Eshuis** (UT), *Semantics and Verification of UML Activity Diagrams for Workflow Modelling.*

2002-16, **Pieter van Langen** (VU), *The Anatomy of Design: Foundations, Models and Applications.*

2002-17, **Stefan Manegold** (UVA), *Understanding, Modeling, and Improving Main-Memory Database Performance.*

2003

2003-01, **Heiner Stuckenschmidt** (VU), *Ontology-Based Information Sharing in Weakly Structured Environments.*

2003-02, **Jan Broersen** (VU), *Modal Action Logics for Reasoning About Reactive Systems.*

2003-03, **Martijn Schuemie** (TUD), *Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy.*

2003-04, **Milan Petkovic** (UT), *Content-Based Video Retrieval Supported by Database Technology.*

2003-05, **Jos Lehmann** (UVA), *Causation in Artificial Intelligence and Law - A modelling approach.*

2003-06, **Boris van Schooten** (UT), *Development and specification of virtual environments.*

2003-07, **Machiel Jansen** (UvA), *Formal Explorations of Knowledge Intensive Tasks.*

2003-08, **Yongping Ran** (UM), *Repair Based Scheduling.*

2003-09, **Rens Kortmann** (UM), *The resolution of visually guided behaviour.*

2003-10, **Andreas Lincke** (UvT), *Electronic Business Negotiation: Some experimental studies on the interaction between medium, innovation context and culture.*

2003-11, **Simon Keizer** (UT), *Reasoning under Uncertainty in Natural Language Dialogue* *using Bayesian Networks.*

2003-12, **Roeland Ordelman** (UT), *Dutch speech recognition in multimedia information retrieval.*

2003-13, **Jeroen Donkers** (UM), *Nosce Hostem - Searching with Opponent Models.*

2003-14, **Stijn Hoppenbrouwers** (KUN), *Freezing Language: Conceptualisation Processes across ICT-Supported Organisations.*

2003-15, **Mathijs de Weerdt** (TUD), *Plan Merging in Multi-Agent Systems.*

2003-16, **Menzo Windhouwer** (CWI), *Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouses.*

2003-17, **David Jansen** (UT), *Extensions of Statecharts with Probability, Time, and Stochastic Timing.*

2003-18, **Levente Kocsis** (UM), *Learning Search Decisions.*

2004

2004-01, **Virginia Dignum** (UU), *A Model for Organizational Interaction: Based on Agents, Founded in Logic.*

2004-02, **Lai Xu** (UvT), *Monitoring Multi-party Contracts for E-business.*

2004-03, **Perry Groot** (VU), *A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving.*

2004-04, **Chris van Aart** (UVA), *Organizational Principles for Multi-Agent Architectures.*

2004-05, **Viara Popova** (EUR), *Knowledge discovery and monotonicity.*

2004-06, **Bart-Jan Hommes** (TUD), *The Evaluation of Business Process Modeling Techniques.*

2004-07, **Elise Boltjes** (UM), *Voorbeeldig onderwijs; voorbeeldgestuurd onderwijs, een opstap naar abstract denken, vooral voor meisjes.*

2004-08, **Joop Verbeek** (UM), *Politie en de Nieuwe Internationale Informatiemarkt, Grensregionale politile gegevensuitwisseling en digitale expertise.*

2004-09, **Martin Caminada** (VU), *For the Sake of the Argument; explorations into argument-based reasoning.*

2004-10, **Suzanne Kabel** (UVA), *Knowledge-rich indexing of learning-objects.*

2004-11, **Michel Klein** (VU), *Change Management for Distributed Ontologies.*

2004-12, **The Duy Bui** (UT), *Creating emotions and facial expressions for embodied agents.*

2004-13, **Wojciech Jamroga** (UT), *Using Multiple Models of Reality: On Agents who Know how to Play.*

2004-14, **Paul Harrenstein** (UU), *Logic in Conflict. Logical Explorations in Strategic Equilibrium.*

2004-15, **Arno Knobbe** (UU), *Multi-Relational Data Mining.*

2004-16, **Federico Divina** (VU), *Hybrid Genetic Relational Search for Inductive Learning.*

2004-17, **Mark Winands** (UM), *Informed Search in Complex Games.*

2004-18, **Vania Bessa Machado** (UvA), *Supporting the Construction of Qualitative Knowledge Models.*

2004-19, **Thijs Westerveld** (UT), *Using generative probabilistic models for multimedia retrieval.*

2004-20, **Madelon Evers** (Nyenrode), *Learning from Design: facilitating multidisciplinary design teams.*

2005

2005-01, **Floor Verdenius** (UVA), *Methodological Aspects of Designing Induction-Based Applications.*

2005-02, **Erik van der Werf** (UM), *AI techniques for the game of Go.*

2005-03, **Franc Grootjen** (RUN), *A Pragmatic Approach to the Conceptualisation of Language.*

2005-04, **Nirvana Meratnia** (UT), *Towards Database Support for Moving Object data.*

2005-05, **Gabriel Infante-Lopez** (UVA), *Two-Level Probabilistic Grammars for Natural Language Parsing.*

2005-06, **Pieter Spronck** (UM), *Adaptive Game AI.*

2005-07, **Flavius Frasincar** (TUE), *Hypermedia Presentation Generation for Semantic Web Information Systems.*

2005-08, **Richard Vdovjak** (TUE), *A Model-driven Approach for Building Distributed Ontology-based Web Applications.*

2005-09, **Jeen Broekstra** (VU), *Storage, Querying and Inferencing for Semantic Web Languages.*

2005-10, **Anders Bouwer** (UVA), *Explaining Behaviour: Using Qualitative Simulation in Interactive Learning Environments.*

2005-11, **Elth Ogston** (VU), *Agent Based Matchmaking and Clustering - A Decentralized Approach to Search.*

2005-12, **Csaba Boer** (EUR), *Distributed Simulation in Industry.*

2005-13, **Fred Hamburg** (UL), *Een Computermodel voor het Ondersteunen van Euthanasiebeslissingen.*

2005-14, **Borys Omelayenko** (VU), *Web-*

Service configuration on the Semantic Web; Exploring how semantics meets pragmatics.

2005-15, **Tibor Bosse** (VU), *Analysis of the Dynamics of Cognitive Processes.*

2005-16, **Joris Graaumans** (UU), *Usability of XML Query Languages.*

2005-17, **Boris Shishkov** (TUD), *Software Specification Based on Re-usable Business Components.*

2005-18, **Danielle Sent** (UU), *Test-selection strategies for probabilistic networks.*

2005-19, **Michel van Dartel** (UM), *Situated Representation.*

2005-20, **Cristina Coteanu** (UL), *Cyber Consumer Law, State of the Art and Perspectives.*

2005-21, **Wijnand Derks** (UT), *Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics.*

2006

2006-01, **Samuil Angelov** (TUE), *Foundations of B2B Electronic Contracting.*

2006-02, **Cristina Chisalita** (VU), *Contextual issues in the design and use of information technology in organizations.*

2006-03, **Noor Christoph** (UVA), *The role of metacognitive skills in learning to solve problems.*

2006-04, **Marta Sabou** (VU), *Building Web Service Ontologies.*

2006-05, **Cees Pierik** (UU), *Validation Techniques for Object-Oriented Proof Outlines.*

2006-06, **Ziv Baida** (VU), *Software-aided Service Bundling - Intelligent Methods & Tools for Graphical Service Modeling.*

2006-07, **Marko Smiljanic** (UT), *XML schema matching – balancing efficiency and effectiveness by means of clustering.*

2006-08, **Eelco Herder** (UT), *Forward, Back and Home Again - Analyzing User Behavior on the Web.*

2006-09, **Mohamed Wahdan** (UM), *Automatic Formulation of the Auditor's Opinion.*

2006-10, **Ronny Siebes** (VU), *Semantic Routing in Peer-to-Peer Systems.*

2006-11, **Joeri van Ruth** (UT), *Flattening Queries over Nested Data Types.*

2006-12, **Bert Bongers** (VU), *Interactivation - Towards an e-cology of people, our technological environment, and the arts.*

2006-13, **Henk-Jan Lebbink** (UU), *Dialogue and Decision Games for Information Exchanging Agents.*

2006-14, **Johan Hoorn** (VU), *Software Requirements: Update, Upgrade, Redesign - towards a Theory of Requirements Change.*

2006-15, **Rainer Malik** (UU), *CONAN: Text Mining in the Biomedical Domain.*

2006-16, **Carsten Riggelsen** (UU), *Approximation Methods for Efficient Learning of Bayesian Networks.*

2006-17, **Stacey Nagata** (UU), *User Assistance for Multitasking with Interruptions on a Mobile Device.*

2006-18, **Valentin Zhizhkun** (UVA), *Graph transformation for Natural Language Processing.*

2006-19, **Birna van Riemsdijk** (UU), *Cognitive Agent Programming: A Semantic Approach.*

2006-20, **Marina Velikova** (UvT), *Monotone models for prediction in data mining.*

2006-21, **Bas van Gils** (RUN), *Aptness on the Web.*

2006-22, **Paul de Vrieze** (RUN), *Fundaments of Adaptive Personalisation.*

2006-23, **Ion Juvina** (UU), *Development of Cognitive Model for Navigating on the Web.*

2006-24, **Laura Hollink** (VU), *Semantic Annotation for Retrieval of Visual Resources.*

2006-25, **Madalina Drugan** (UU), *Conditional log-likelihood MDL and Evolutionary MCMC.*

2006-26, **Vojkan Mihajlovic** (UT), *Score Region Algebra: A Flexible Framework for Structured Information Retrieval.*

2006-27, **Stefano Bocconi** (CWI), *Vox Populi: generating video documentaries from semantically annotated media repositories.*

2006-28, **Borkur Sigurbjornsson** (UVA), *Focused Information Access using XML Element Retrieval.*

2007

2007-01, **Kees Leune** (UvT), *Access Control and Service-Oriented Architectures.*

2007-02, **Wouter Teepe** (RUG), *Reconciling Information Exchange and Confidentiality: A Formal Approach.*

2007-03, **Peter Mika** (VU), *Social Networks and the Semantic Web.*

2007-04, **Jurriaan van Diggelen** (UU), *Achieving Semantic Interoperability in Multi-agent Systems: a dialogue-based approach.*

2007-05, **Bart Schermer** (UL), *Software Agents, Surveillance, and the Right to Privacy: a Legislative Framework for Agent-enabled Surveillance.*

2007-06, **Gilad Mishne** (UVA), *Applied Text Analytics for Blogs.*

2007-07, **Natasa Jovanovic'** (UT), *To Whom It May Concern - Addressee Identification in Face-to-Face Meetings.*

2007-08, **Mark Hoogendoorn** (VU), *Modeling* *of Change in Multi-Agent Organizations.*

2007-09, **David Mobach** (VU), *Agent-Based Mediated Service Negotiation.*

2007-10, **Huib Aldewereld** (UU), *Autonomy vs. Conformity: an Institutional Perspective on Norms and Protocols.*

2007-11, **Natalia Stash** (TUE), *Incorporating Cognitive/Learning Styles in a General-Purpose Adaptive Hypermedia System.*

2007-12, **Marcel van Gerven** (RUN), *Bayesian Networks for Clinical Decision Support: A Rational Approach to Dynamic Decision-Making under Uncertainty.*

2007-13, **Rutger Rienks** (UT), *Meetings in Smart Environments; Implications of Progressing Technology.*

2007-14, **Niek Bergboer** (UM), *Context-Based Image Analysis.*

2007-15, **Joyca Lacroix** (UM), *NIM: a Situated Computational Memory Model.*

2007-16, **Davide Grossi** (UU), *Designing Invisible Handcuffs. Formal investigations in Institutions and Organizations for Multi-agent Systems.*

2007-17, **Theodore Charitos** (UU), *Reasoning with Dynamic Networks in Practice.*

2007-18, **Bart Orriens** (UvT), *On the development an management of adaptive business collaborations.*

2007-19, **David Levy** (UM), *Intimate relationships with artificial partners.*

2007-20, **Slinger Jansen** (UU), *Customer Configuration Updating in a Software Supply Network.*

2007-21, **Karianne Vermaas** (UU), *Fast diffusion and broadening use: A research on residential adoption and usage of broadband internet in the Netherlands between 2001 and 2005.*

2007-22, **Zlatko Zlatev** (UT), *Goal-oriented design of value and process models from patterns.*

2007-23, **Peter Barna** (TUE), *Specification of Application Logic in Web Information Systems.*

2007-24, **Georgina Ramrez Camps** (CWI), *Structural Features in XML Retrieval.*

2007-25, **Joost Schalken** (VU), *Empirical Investigations in Software Process Improvement.*

2008

2008-01, **Katalin Boer-Sorbn** (EUR), *Agent-Based Simulation of Financial Markets: A modular, continuous-time approach.*

2008-02, **Alexei Sharpanskykh** (VU), *On Computer-Aided Methods for Modeling and Analysis of Organizations.*

2008-03, **Vera Hollink** (UVA), *Optimizing hierarchical menus: a usage-based approach.*

2008-04, **Ander de Keijzer** (UT), *Management of Uncertain Data - towards unattended integration.*

2008-05, **Bela Mutschler** (UT), *Modeling and simulating causal dependencies on process-aware information systems from a cost perspective.*

2008-06, **Arjen Hommersom** (RUN), *On the Application of Formal Methods to Clinical Guidelines, an Artificial Intelligence Perspective.*

2008-07, **Peter van Rosmalen** (OU), *Supporting the tutor in the design and support of adaptive e-learning.*

2008-08, **Janneke Bolt** (UU), *Bayesian Networks: Aspects of Approximate Inference.*

2008-09, **Christof van Nimwegen** (UU), *The paradox of the guided user: assistance can be counter-effective.*

2008-10, **Wauter Bosma** (UT), *Discourse oriented summarization.*

2008-11, **Vera Kartseva** (VU), *Designing Controls for Network Organizations: A Value-Based Approach.*

2008-12, **Jozsef Farkas** (RUN), *A Semiotically Oriented Cognitive Model of Knowledge Representation.*

2008-13, **Caterina Carraciolo** (UVA), *Topic Driven Access to Scientific Handbooks.*

2008-14, **Arthur van Bunningen** (UT), *Context-Aware Querying; Better Answers with Less Effort.*

2008-15, **Martijn van Otterlo** (UT), *The Logic of Adaptive Behavior: Knowledge Representation and Algorithms for the Markov Decision Process Framework in First-Order Domains..*

2008-16, **Henriette van Vugt** (VU), *Embodied agents from a user's perspective.*

2008-17, **Martin Op 't Land** (TUD), *Applying Architecture and Ontology to the Splitting and Allying of Enterprises.*

2008-18, **Guido de Croon** (UM), *Adaptive Active Vision.*

2008-19, **Henning Rode** (UT), *From Document to Entity Retrieval: Improving Precision and Performance of Focused Text Search.*

2008-20, **Rex Arendsen** (UVA), *Geen bericht, goed bericht. Een onderzoek naar de effecten van de introductie van elektronisch berichtenverkeer met de overheid op de administratieve lasten van bedrijven..*

2008-21, **Krisztian Balog** (UVA), *People Search in the Enterprise.*

2008-22, **Henk Koning** (UU), *Communication of IT-Architecture.*

2008-23, **Stefan Visscher** (UU), *Bayesian network models for the management of ventilator-associated pneumonia.*

2008-24, **Zharko Aleksovski** (VU), *Using background knowledge in ontology matching.*

2008-25, **Geert Jonker** (UU), *Efficient and Equitable Exchange in Air Traffic Management Plan Repair using Spender-signed Currency.*

2008-26, **Marijn Huijbregts** (UT), *Segmentation, Diarization and Speech Transcription: Surprise Data Unraveled.*

2008-27, **Hubert Vogten** (OU), *Design and Implementation Strategies for IMS Learning Design.*

2008-28, **Ildiko Flesch** (RUN), *On the Use of Independence Relations in Bayesian Networks.*

2008-29, **Dennis Reidsma** (UT), *Annotations and Subjective Machines - Of Annotators, Embodied Agents, Users, and Other Humans.*

2008-30, **Wouter van Atteveldt** (VU), *Semantic Network Analysis: Techniques for Extracting, Representing and Querying Media Content.*

2008-31, **Loes Braun** (UM), *Pro-Active Medical Information Retrieval.*

2008-32, **Trung H. Bui** (UT), *Toward Affective Dialogue Management using Partially Observable Markov Decision Processes.*

2008-33, **Frank Terpstra** (UVA), *Scientific Workflow Design; theoretical and practical issues.*

2008-34, **Jeroen de Knijf** (UU), *Studies in Frequent Tree Mining.*

2008-35, **Ben Torben Nielsen** (UvT), *Dendritic morphologies: function shapes structure.*

2009

2009-01, **Rasa Jurgelenaite** (RUN), *Symmetric Causal Independence Models.*

2009-02, **Willem Robert van Hage** (VU), *Evaluating Ontology-Alignment Techniques.*

2009-03, **Hans Stol** (UvT), *A Framework for Evidence-based Policy Making Using IT.*

2009-04, **Josephine Nabukenya** (RUN), *Improving the Quality of Organisational Policy Making using Collaboration Engineering.*

2009-05, **Sietse Overbeek** (RUN), *Bridging Supply and Demand for Knowledge Intensive Tasks - Based on Knowledge, Cognition, and Quality.*

2009-06, **Muhammad Subianto** (UU), *Understanding Classification.*

2009-07, **Ronald Poppe** (UT), *Discriminative Vision-Based Recovery and Recognition of Human Motion.*

2009-08, **Volker Nannen** (VU), *Evolutionary Agent-Based Policy Analysis in Dynamic Environments.*

2009-09, **Benjamin Kanagwa** (RUN), *Design, Discovery and Construction of Service-oriented Systems.*

2009-10, **Jan Wielemaker** (UVA), *Logic programming for knowledge-intensive interactive applications.*

2009-11, **Alexander Boer** (UVA), *Legal Theory, Sources of Law & the Semantic Web.*

2009-12, **Peter Massuthe** (TUE, Humboldt-Universitaet zu Berlin), *Operating Guidelines for Services.*

2009-13, **Steven de Jong** (UM), *Fairness in Multi-Agent Systems.*

2009-14, **Maksym Korotkiy** (VU), *From ontology-enabled services to service-enabled ontologies (making ontologies work in e-science with ONTO-SOA).*

2009-15, **Rinke Hoekstra** (UVA), *Ontology Representation - Design Patterns and Ontologies that Make Sense.*

2009-16, **Fritz Reul** (UvT), *New Architectures in Computer Chess.*

2009-17, **Laurens van der Maaten** (UvT), *Feature Extraction from Visual Data.*

2009-18, **Fabian Groffen** (CWI), *Armada, An Evolving Database System.*

2009-19, **Valentin Robu** (CWI), *Modeling Preferences, Strategic Reasoning and Collaboration in Agent-Mediated Electronic Markets.*

2009-20, **Bob van der Vecht** (UU), *Adjustable Autonomy: Controling Influences on Decision Making.*

2009-21, **Stijn Vanderlooy** (UM), *Ranking and Reliable Classification.*

2009-22, **Pavel Serdyukov** (UT), *Search For Expertise: Going beyond direct evidence.*

2009-23, **Peter Hofgesang** (VU), *Modelling Web Usage in a Changing Environment.*

2009-24, **Annerieke Heuvelink** (VUA), *Cognitive Models for Training Simulations.*

2009-25, **Alex van Ballegooij** (CWI), *"RAM: Array Database Management through Relational Mapping".*

2009-26, **Fernando Koch** (UU), *An Agent-Based Model for the Development of Intelligent Mobile Services.*

2009-27, **Christian Glahn** (OU), *Contextual Support of social Engagement and Reflection on the Web.*

2009-28, **Sander Evers** (UT), *Sensor Data Management with Probabilistic Models.*

2009-29, **Stanislav Pokraev** (UT), *Model-Driven Semantic Integration of Service-Oriented Applications.*

2009-30, **Marcin Zukowski** (CWI), *Balancing vectorized query execution with bandwidth-optimized storage.*

2009-31, **Sofiya Katrenko** (UVA), *A Closer*

Look at Learning Relations from Text.

2009-32, **Rik Farenhorst (VU) and Remco de Boer** (VU), *Architectural Knowledge Management: Supporting Architects and Auditors.*

2009-33, **Khiet Truong** (UT), *How Does Real Affect Affect Affect Recognition In Speech?.*

2009-34, **Inge van de Weerd** (UU), *Advancing in Software Product Management: An Incremental Method Engineering Approach.*

2009-35, **Wouter Koelewijn** (UL), *Privacy en Politiegegevens; Over geautomatiseerde normatieve informatie-uitwisseling.*

2009-36, **Marco Kalz** (OUN), *Placement Support for Learners in Learning Networks.*

2009-37, **Hendrik Drachsler** (OUN), *Navigation Support for Learners in Informal Learning Networks.*

2009-38, **Riina Vuorikari** (OU), *Tags and self-organisation: a metadata ecology for learning resources in a multilingual context.*

2009-39, **Christian Stahl** (TUE, Humboldt-Universitaet zu Berlin), *Service Substitution – A Behavioral Approach Based on Petri Nets.*

2009-40, **Stephan Raaijmakers** (UvT), *Multinomial Language Learning: Investigations into the Geometry of Language.*

2009-41, **Igor Berezhnyy** (UvT), *Digital Analysis of Paintings.*

2009-42, **Toine Bogers** (UvT), *Recommender Systems for Social Bookmarking.*

2009-43, **Virginia Nunes Leal Franqueira** (UT), *Finding Multi-step Attacks in Computer Networks using Heuristic Search and Mobile Ambients.*

2009-44, **Roberto Santana Tapia** (UT), *Assessing Business-IT Alignment in Networked Organizations.*

2009-45, **Jilles Vreeken** (UU), *Making Pattern Mining Useful.*

2009-46, **Loredana Afanasiev** (UvA), *Querying XML: Benchmarks and Recursion.*

2010

2010-01, **Matthijs van Leeuwen** (UU), *Patterns that Matter.*

2010-02, **Ingo Wassink** (UT), *Work flows in Life Science.*

2010-03, **Joost Geurts** (CWI), *A Document Engineering Model and Processing Framework for Multimedia documents.*

2010-04, **Olga Kulyk** (UT), *Do You Know What I Know? Situational Awareness of Co-located Teams in Multidisplay Environments.*

2010-05, **Claudia Hauff** (UT), *Predicting the Effectiveness of Queries and Retrieval Systems.*

2010-06, **Sander Bakkes** (UvT), *Rapid Adaptation of Video Game AI.*

2010-07, **Wim Fikkert** (UT), *Gesture interaction at a Distance.*

2010-08, **Krzysztof Siewicz** (UL), *Towards an Improved Regulatory Framework of Free Software. Protecting user freedoms in a world of software communities and eGovernments.*

2010-09, **Hugo Kielman** (UL), *A Politiele gegevensverwerking en Privacy, Naar een effectieve waarborging.*

2010-10, **Rebecca Ong** (UL), *Mobile Communication and Protection of Children.*

2010-11, **Adriaan Ter Mors** (TUD), *The world according to MARP: Multi-Agent Route Planning.*

2010-12, **Susan van den Braak** (UU), *Sensemaking software for crime analysis.*

2010-13, **Gianluigi Folino** (RUN), *High Performance Data Mining using Bio-inspired techniques.*

2010-14, **Sander van Splunter** (VU), *Automated Web Service Reconfiguration.*

2010-15, **Lianne Bodenstaff** (UT), *Managing Dependency Relations in Inter-Organizational Models.*

2010-16, **Sicco Verwer** (TUD), *Efficient Identification of Timed Automata, theory and practice.*

2010-17, **Spyros Kotoulas** (VU), *Scalable Discovery of Networked Resources: Algorithms, Infrastructure, Applications.*

2010-18, **Charlotte Gerritsen** (VU), *Caught in the Act: Investigating Crime by Agent-Based Simulation.*

2010-19, **Henriette Cramer** (UvA), *People's Responses to Autonomous and Adaptive Systems.*

2010-20, **Ivo Swartjes** (UT), *Whose Story Is It Anyway? How Improv Informs Agency and Authorship of Emergent Narrative.*

2010-21, **Harold van Heerde** (UT), *Privacy-aware data management by means of data degradation.*

2010-22, **Michiel Hildebrand** (CWI), *End-user Support for Access to Heterogeneous Linked Data.*

2010-23, **Bas Steunebrink** (UU), *The Logical Structure of Emotions.*

2010-24, **Dmytro Tykhonov** (TUD), *Designing Generic and Efficient Negotiation Strategies.*

2010-25, **Zulfiqar Ali Memon** (VU), *Modelling Human-Awareness for Ambient Agents: A Human Mindreading Perspective.*

2010-26, **Ying Zhang** (CWI), *XRPC: Efficient Distributed Query Processing on Heterogeneous XQuery Engines.*

2010-27, **Marten Voulon** (UL), *Automatisch* *contracteren.*

2010-28, **Arne Koopman** (UU), *Characteristic Relational Patterns.*

2010-29, **Stratos Idreos** (CWI), *Database Cracking: Towards Auto-tuning Database Kernels.*

2010-30, **Marieke van Erp** (UvT), *Accessing Natural History - Discoveries in data cleaning, structuring, and retrieval.*

2010-31, **Victor de Boer** (UVA), *Ontology Enrichment from Heterogeneous Sources on the Web.*

2010-32, **Marcel Hiel** (UvT), *An Adaptive Service Oriented Architecture: Automatically solving Interoperability Problems.*

2010-33, **Robin Aly** (UT), *Modeling Representation Uncertainty in Concept-Based Multimedia Retrieval.*

2010-34, **Teduh Dirgahayu** (UT), *Interaction Design in Service Compositions.*

2010-35, **Dolf Trieschnigg** (UT), *Proof of Concept: Concept-based Biomedical Information Retrieval.*

2010-36, **Jose Janssen** (OU), *Paving the Way for Lifelong Learning; Facilitating competence development through a learning path specification.*

2010-37, **Niels Lohmann** (TUE), *Correctness of services and their composition.*

2010-38, **Dirk Fahland** (TUE), *From Scenarios to components.*

2010-39, **Ghazanfar Farooq Siddiqui** (VU), *Integrative modeling of emotions in virtual agents.*

2010-40, **Mark van Assem** (VU), *Converting and Integrating Vocabularies for the Semantic Web.*

2010-41, **Guillaume Chaslot** (UM), *Monte-Carlo Tree Search.*

2010-42, **Sybren de Kinderen** (VU), *Needs-driven service bundling in a multi-supplier setting - the computational e3-service approach.*

2010-43, **Peter van Kranenburg** (UU), *A Computational Approach to Content-Based Retrieval of Folk Song Melodies.*

2010-44, **Pieter Bellekens** (TUE), *An Approach towards Context-sensitive and User-adapted Access to Heterogeneous Data Sources, Illustrated in the Television Domain.*

2010-45, **Vasilios Andrikopoulos** (UvT), *A theory and model for the evolution of software services.*

2010-46, **Vincent Pijpers** (VU), *e3alignment: Exploring Inter-Organizational Business-ICT Alignment.*

2010-47, **Chen Li** (UT), *Mining Process Model Variants: Challenges, Techniques, Examples.*

2010-48, **Milan Lovric** (EUR), *Behavioral Finance and Agent-Based Artificial Markets.*

2010-49, **Jahn-Takeshi Saito** (UM), *Solving difficult game positions.*

2010-50, **Bouke Huurnink** (UVA), *Search in Audiovisual Broadcast Archives.*

2010-51, **Alia Khairia Amin** (CWI), *Understanding and supporting information seeking tasks in multiple sources.*

2010-52, **Peter-Paul van Maanen** (VU), *Adaptive Support for Human-Computer Teams: Exploring the Use of Cognitive Models of Trust and Attention.*

2010-53, **Edgar Meij** (UVA), *Combining Concepts and Language Models for Information Access.*