# Using Event Calculus for Normative Reasoning of BDI agents in the context of Norm Identification

by

© *Wagdi Alrawagfeh*

A thesis submitted to the

School of Graduate Studies

in partial fulfillment of the requirements for the degree of

**Doctor of Philosophy**

**Department of Computer Science**

Memorial University of Newfoundland

*May 2015*

St. John's Newfoundland

# Abstract

In human society, where different backgrounds, cultures and objectives coexist, norms help predict, control and coordinate individual behavior. Similarly, norms have been used in multi-agent systems to describe ideal behaviour for software agents. In spite of this, agents are still expected behave autonomously, due to the leeway allowed by norms as soft constraints on individual behavior. Existing work dealing with norm identification in multi-agent systems generally assumes that agents are fully aware of all norms, either at design time or as a result of communication with other agents. Similarly, work examining the impact of norms in agent decision-making proposes strategies that assume agents have complete knowledge of normative states.

This thesis proposes that agents do not have complete knowledge about normative states; consequently, it is the agents' duty to identify norms. To this end, we propose an agent architecture and algorithms for identifying dynamic permission and prohibition norms in open multi-agent systems. Using Event Calculus, we propose a formal representation of norms and a normative practical reasoning mechanism. Other studies assume that the normative states that are neither identified prohibited nor obliged are permitted. Central to our proposal is that a normative state can be unknown if it is not explicitly identified as prohibited, obliged or permitted. This allows us to integrate permission norms into our proposed normative practical reasoning mechanism. Thus, the contribution of this thesis is a set of techniques and algorithms that allow agents to join and function in a society regulated by (possibly unknown) norms, while minimizing behaviour that violates such norms.

## Acknowledgements

I would like to thank my supervisor Dr. Jian Tang and Dr. Mohamed Hossam for their help and encouragement. Their support will not be forgotten.

I also would like to thank Dr. Edward Brown and Dr. Manrique Mata-Montero for their help at the first part of my research. I really appreciate the time we spent together pondering and discussing my research. I learned several things from Dr. Brown such as, working hard, being helpful and how to unravel sense from nonsense issues. I learned several things from Dr. Mata-Montero as well, mainly, how to think deep, be always aware of human language ambiguity and questioning everything.

My special appreciation going to Dr. Felipe Meneguzzi, I do not find enough words to express my appreciation to him. Simply, without Dr. Meneguzzi's help this research would not be possible. He was and still is helping me generously and answering my endless questions. I am fortunate to have a wonderful committee member like Dr. Meneguzzi who has been extremely supportive and encouraging during my study.

I also would like to thank a great professor who has a big heart and provides a caring atmosphere for his students, Dr. Todd Wareham.

Appreciation goes out to the staff and faculty of the Computer Science Department who offered me assistance in many levels of my graduate studies, special thanks to Dr. Krishnamurthy Vidyasankar, Dr. Tina Gwoing Yu, Dr. Yuanzhu Peter Chen and Dr. Wolfgang Banzhaf.

I would like to thank Darlene Oliver, Brenda Hiller, Sharon Deir, Elaine Boone, Nolan White, and Paul Price for their help and support during completion of my pro-

gram. I also acknowledge the scholarship of the Tafila Technical University, Jordan (TTU) and the financial contributions of the School of Graduate Studies of Memorial University which supported my graduate studies. I would like also to thank Mr.Andrew Kim from the School or graduate studies, Mrs.Juanita Hennesey from the International Student Advising Office and Dr. Carrie Dyck and Sarah Knee from the Department of Linguistics for their help and support. Thanks to the fellow students in the department of Computer Science for their friendship and support. Thanks to Kais Laribi, Ranjeet Kumar and Wasiq Waqar for our great time and beautiful memories. Thanks to my friends, Dr. Ahmad Zain, Dr. Khalid Omari, Dr. Eid Alna'emat and Ahmad Fahd, they were always my source of support and stress-relief during my study trip. Thanks to my old friends Mohamad Khader Alzomor and Dr. Raed Abu Zitar, their pure words and encouragement to pursue the Ph.D stuck on my mind.

# Dedication

All praise to ALLAH who covers me of his blessings all my life. My greatest gratefulness go to my beloved mother , Mariam, and to the soul of my father, Mahmoud, may ALLAH bless his soul. I also would like to thank my brothers and sisters especially my brother, Sami, for giving me a strong faith and inspiration to complete my Ph.D. I also would like to thank my uncle, Abu Talal, for his support. Especial gratitude goes to the brightness of my life, my wife, Eman, and my beloved children: Shahd, Habeeb, Jana, Huda and Fatima. They were extremely supportive and patient during my Ph.D journey.

# Contents

# List of Algorithms

# List of Tables

# List of Figures

# Chapter 1

# Introduction

The rapid development of information technology has created the problem of finding new system software paradigms that can cope with the pace and nature of this development. For example, writing computer programs for fixed tasks is no longer suitable in light of the large, distributed, complex and dynamic environments in which such programs must function. For such systems, where features like parallelism, robustness, scalability and simpler programming are needed, multi-agent systems offer a solution (Stone and Veloso, 2000).

Predefined rules that govern how a system or an agent behave are not adequate mechanisms especially in complex and dynamic environments. In such systems, norms have been used as a standard changeable specification of desirable agent behaviour. For more than a decade the norm concept has been utilized in multi-agent systems (Hollander and Wu, 2011). In this thesis, we study two sub-fields in normative multi-agent systems: norm identification and normative practical reasoning.

Before going into our research, we provide an introduction to the field of multi-agent systems in Section 1.1. In Section 1.2, we explain the motivation for our work and summarize our contributions to the field. We end this chapter outlining the main objectives of this thesis in Section 1.3.

## 1.1 Multi-agent systems

According to Wooldridge (2002), agents are software components situated so that they can observe the environment as well as make decisions and perform actions that affect the environment and their own or other agents' internal states. Software agents are distinguished from traditional software programs by the following properties:

1. Autonomy: autonomous agents have control over their internal states and behaviour and can operate without direct intervention from human or other agents.

2. Perceptiveness: agents are able to perceive the changes that may happen in the environment and react to those changes in a timely fashion.

3. Proactivity: agents do not simply act in response to their environment, but introduce goal-directed behaviours by taking initiatives.

4. Social interactivity: agents can communicate with each other using some kind of agent communication language. Based on the context, agents may compete, cooperate or negotiate.

According to (Hayzelden et al., 1999; Wooldridge and Jennings, 1994) agent architecture is a fundamental characteristic of an agent which helps it to present effective behaviour in a dynamic and open environment. Basically, agent architecture

describes agent's modules and how these modules work together. Agent architectures have been classified into *reactive*, *proactive*, *hybrid* architectures. *Reactive* architecture is one of the simplest architecture concentrates on reactivity based on behavioral rules. A behavioral rule is triggered as a response to inputs. These kinds of architectures have a limited representation of the environment and use limited reasoning. They focus on a quick response to the detected environment's changes. *Deliberative* architectures are based on long-term planning of actions and symbolic reasoning and they have a richer representation of the environment. Actions in *deliberative* architectures are taken based on logical reasoning. In spite of the usefulness and expressive power of these kinds of architectures in many domains, they are still limited to computational intractability. Researchers suggested that neither reactive nor deliberative architecture is suitable to build a software agent (Wooldridge and Jennings, 1994). *Hybrid* architecture came as a balance between the reactive and deliberative architecture. BDI (beliefs, desires and intentions) architecture (Bratman, 1987) is a hybrid architecture based on mental states in deciding its actions (see Section 2.2).

A group of interacting agents form a multi-agent system. Multi-agent systems form a promising software paradigm for building open information systems, mainly because of the possibility of using multi-agent systems protocols that coordinate agents interoperability (Wooldridge, 2002). A closed multi-agent system is composed of homogeneous agents where agents have identical design and are programmed to achieve common objectives. In such systems, agents can exchange information and services without difficulties, where specific interactions can be hard-coded at design time (Dellarocas and Klein, 2001; Dignum et al., 2005). In contrast, an open multi-agent system is a complex system, meaning that it is composed of several heterogeneous agents that can have joint or self-interested objectives. Furthermore, agents in open multi-agent

systems may be designed by different owners, and agents may join and leave the system autonomously (Huynh et al., 2006). Research on multi-agent systems is devoted to taming such complexity and to nurturing coordination among agents, while at the same time preserving agents' autonomy.

Researchers in multi-agent systems adopt concepts from several disciplines. For example, concepts from philosophy such as human behaviours are characterized based on a model of human beliefs, desires and intentions (Bratman, 1987). Explicitly representing these mental states in agent architecture produces agents whose behaviours are affected by the adopted beliefs, desires and intentions (see Section 2.2). Another example is the concept of norms from social science. Norms govern the behaviours of individuals in a society (Jones and Sergot, 1993). Accordingly, researchers in multi-agent systems adopt the concept of norm to imitate the collective behaviours of human society.

## 1.2 Motivation and contribution

In a multi-agent society regulated by norms, agents may join the society in order to achieve their own objectives. When agents exhibit undesirable behaviours such as ignoring or violating the norms of the joined society, such behaviours may result in the loss of resources (time, money, scores, etc.) from the perspective of an individual agent. To avoid such behaviours, agents need to know the norms of the joined societies and to take them into consideration during their practical reasoning.

Several models have been proposed to implement dynamic norm awareness behaviour. Boman (1999) assumes that norms are established by a social authority or

a legislator, while others state that norms could emerge from the society or be negotiated among agents (see for example Boella and Van Der Torre, 2007b; Andrighetto et al., 2008; Campenní et al., 2009). While many researchers agree that agents need to infer norms in certain environments (Savarimuthu, 2011; Andrighetto et al., 2010; Mahmoud et al., 2012b,a; Oren and Meneguzzi, 2013; Savarimuthu et al., 2013), there is no consensus with respect to the approach for norm identification.

Once agents learn norms, the question becomes how norms affect agent decision-making. For that purpose, several models study the impact of norms in agents' practical reasoning (e.g., Kollingbaum, 2005; Meneguzzi and Luck, 2009; Oren et al., 2011; Alechina et al., 2012; Criado et al., 2011; Balke et al., 2012; Panagiotidi and Vázquez-Salceda, 2012; Meneguzzi et al., 2012). These models of normative practical reasoning all assume that an agent has complete knowledge about the normative states of a system. Accordingly, they assume that whatever is not prohibited is permitted. However, this is not necessarily true, particularly in uncertain environments and situations with newcomer agents; in such cases, for example, if an agent does not know whether action `A` is prohibited, it does not necessarily know that action `A` is permitted in the environment.

The decision making model underlying the BDI agent is known as practical reasoning. Practical reasoning is the reasoning directed towards actions. While there has been work focusing on normative practical reasoning and on norm identification, we are not aware of any work that has studied these two problems together. Studying these two problems separately results in a gap between the expressive power of norms at the norm identification level and at the normative practical reasoning level. Researchers who work in normative practical reasoning assume that agents already know norms beforehand. Therefore, they have more flexibility in forming their view

of norms and adding details to the representation of norms.

To be more realistic, we relax the assumption that norms are known beforehand. To do this we delegate the duty of detecting norms to the agent itself. In our view, it is the duty of the agent to detect norms and adapt its own behaviour to comply with such norms. Consequently, the expressive power of norms that results from the norm identification process should be compatible with norm representations that are used in the practical reasoning process. In other words, the normative reasoning process should be based on the expressivity of the underlying norms. Further, because norms are determined by the capabilities of the norm identification process, normative practical reasoning and the norm identification should not be studied separately.

In our work we suppose that the multi-agent society is already established; it has norms, agents and norm enforcement authority. Agents and the norm enforcement authority are aware of the prevailing norms. Our agent (the newcomer agent) who joins the system is not aware of the prevailing norms. In our experiments, we suppose that all agents are in the recognizer agent's vicinity.

In order to incorporate norms into an agent's practical reasoning, norms need to be represented in a formal way. Agents need to know the effects of their actions, taking norms into account. Since event calculus (Kowalski and Sergot, 1989) is concerned with reasoning about actions and their effects, it is a suitable language for representing norms. In addition to the simplicity of representing concepts in event calculus, the resulting representation is written in Horn clauses (Horn, 1951), which in turn are directly executable in logic programming languages such as Prolog or Jason.

To summarize, our model makes the following contributions:

- We set a framework to establish building a BDI-agent able to identify norms and use them in its normative practical reasoning.
- We propose an agent architecture and mechanisms for detecting new and repealed norms in an open multi-agent society.
- We extend classical event calculus to propose a formal representation of obligation, prohibition and permission norms.
- Using event calculus, we propose a mechanism to take norms into account during agents' practical reasoning.
- We introduce the idea of inferring permission norms by observing regular (non-sanction) events.
- We propose the idea of utilizing permission norms in detecting repealed prohibition norms and in enhancing BDI-agent practical reasoning.

This thesis differs from the earlier work in four important ways:

1. Our norm identification mechanism identifies permission norms. Identifying permission norms is significant since it gives agents the ability to discover repealed prohibition norms, hence, identifying changeable norms. For example, using a traffic system analogy, an agent A does not remove the prohibition against running red lights until it reclassifies the norm as a permission norm through observing that several incidents of running red lights occur without punishment.
2. In our norm identification mechanism, we avoid the need to communicate with other agents using the concept of norm. Instead we use the Foundation for

Intelligent Physical Agents (FIPA)'s Contract Net Interaction Protocol (FIPA, 2002b). As a result we avoid direct communications about norms with other agents, and we avoid transmitting potentially misleading information obtained from other agents in open multi-agent societies.

3. In our agent practical reasoning mechanism, we assume that agents do not know norms beforehand. It is the agents' duty to infer the norms within a society. This assumption has at least two consequences:

   (a) The expressive power of norms should be compatible with the type of norms that can be inferred. Therefore, we take norm identification into account when we design the normative practical reasoning mechanism.

   (b) Agents do not have a complete knowledge about the normative states of a system (what is prohibited, permitted or obligatory). This could be for several reasons, such as imperfection of the norm identification mechanism or because norms by their nature are changeable. As a result, our proposed normative practical reasoning mechanism can operate in environments where there is uncertainty about normative states.

   Our agent practical reasoning mechanism not only reasons about current actions that an agent is about to perform, but also reasons about the combination of current actions and previously performed actions.

4. Since most researchers adopt the Sealing Principle (Royakkers, 1997), which states that "whatever is not prohibited is permitted", permission norms have not been fully exploited in normative reasoning. The sealing principle is sound if agents have complete knowledge about the normative states of a particular system so they can always determine whether some action is prohibited or per-

mitted. However, in this thesis we assume that agents do not have complete knowledge, so some actions will be known as either prohibited, obliged, or permitted, whereas the status of others will be unknown. As we relax the sealing principle assumption, we argue that permission norms have a significant role in normative agents' practical reasoning.

## 1.3 Objectives and organization

The main objective of this thesis is to establish a framework and define algorithms for creating a BDI agent capable of joining and functioning in a society regulated by (possibly unknown) norms, while minimizing behaviour that violates norms.
To achieve this objective, the following research questions are investigated:

1. How are new norms and repealed norms identified in open multi-agent societies?
2. What mechanisms do agents need in order to change their behaviour to avoid norm violation?

This thesis is organized as follows: First we provide an overview of norm and norm representation, and a background of the technologies we use in our work (Chapter 2). In Chapters 3 and 4, we present an agent architecture together with algorithms for identifying prohibition and permission norms in open multi-agent systems. In the next two chapters (5 and 6) we propose a formal representation of norms and a mechanism for integrating norms into BDI-agent practical reasoning. First we propose representations for prohibition and obligation norms and demonstrate how agents are able to find and follow best plans among applicable plans. Then we propose a rep-

resentation for permission norms and demonstrate how agents are able to choose the safest plan from possible best plans. Finally, we summarize our work in Chapter 7.

## 1.4 Resulting publications

The work in this thesis has generated several peer-reviewed publications. These are listed below with reference to their associated chapters. I am the main author for all of these publications, and I made the major intellectual and practical contribution to all work that is reported in this thesis.

As a result of our work in this thesis, we have the following publications:
In Chapters 3 and 4, we present an agent architecture and algorithms for identifying prohibition and permission norms in open multi-agent systems. These Chapters consist of three published papers; Alrawagfeh et al. (2011a) published in a peer-reviewed conference, the 7th Conference of the European Social Simulation Association (ESSA 2011). Alrawagfeh et al. (2011b) published in a peer-reviewed workshop, the Agent-Directed Simulation workshop (ADS) at the Society for Computer Simulation International (SCS 2011), and a journal paper (Alrawagfeh et al., 2011c) published in the International Journal of Agent Technologies and Systems (IJATS @ IGI Global) Volume 3: 2011.

In Chapters 5 and 6, we propose a formal representation of norms and a mechanism for integrating norms into BDI-agent practical reasoning. These Chapters consist of two published papers; (Alrawagfeh, 2013) was published in the Springer LNCS proceedings of a peer-reviewed conference, the 16th International Conference on Principles and Practice of Multi-agent Systems (PRIMA 2013). Alrawagfeh and

Meneguzzi (2015) will appear in the Springer LNCS proceedings of a peer-reviewed workshop, the 17th International Workshop on Coordination, Organizations, Institutions and Norms (COIN) at Autonomous Agents and Multi-agent Systems (AAMAS) conference.

In the next page, we list all of the citation information from the publications resulting from this thesis.

Alrawagfeh, W. (2013). Norm representation and reasoning: A formalization in event calculus. In Boella, G., Elkind, E., Savarimuthu, B., Dignum, F., and Purvis, M., editors, *PRIMA 2013: Principles and Practice of Multi-Agent Systems*, volume 8291 of *Lecture Notes in Computer Science*, pages 5–20, Berlin, Germany. Springer [Chapters 5 and 6].

Alrawagfeh, W., Brown, E., and Mata-Mantero, M. (2011a). Identifying norms of behaviour in multi-agent societies. In *the 7th Conference of the European Social Simulation Association (ESSA)*. [Chapters 3 and 4].

Alrawagfeh, W., Brown, E., and Mata-Mantero, M. (2011b). Identifying norms of behaviour in open multi-agent societies. In *Proceedings of the 2011 Workshop on Agent-Directed Simulation*, pages 13–20, Boston, USA. Society for Computer Simulation International [Chapters 3 and 4].

Alrawagfeh, W., Brown, E., and Mata-Montero, M. (2011c). Norms of behaviour and their identification and verification in open multi-agent societies. *International Journal of Agent Technologies and Systems (IJATS)*, 3(3):1–16. [Chapters 3 and 4].

Alrawagfeh, W. and Meneguzzi, F. (2015). Utilizing permission norms in BDI practical normative reasoning. In *16th International Workshop on Coordination, Organizations, Institutions, and Norms*, Lecture Notes in Computer Science, Berlin, Germany. Springer [Chapters 5 and 6].

# Chapter 2

# Background

This chapter presents our view of norm and an introduction to the belief, desire and intention (BDI) agent (Bratman, 1987), in addition to three technologies we use in this thesis. The first is event calculus (EC) (Kowalski and Sergot, 1989), a logical framework for representing and reasoning about actions and their effects. As we study the effects of agent's actions in the presence of norms, event calculus is a suitable formalism for our work. In addition to the simplicity of representing concepts in event calculus, the resulting representation is written in Horn clauses, which in turn are directly executable in logic programming languages such as Prolog or Jason (Bordini and Hübner, 2006).

The second technology we use is Jason, a BDI-based agent programming language. Jason is one of the most popular and active agent BDI programming languages in multi-agent systems. As Jason uses logic programming constructs that allow Prolog-like logical rules in agents definition, it complies with our need of implementing our event calculus formalisms.

JADE is the third technology we use in our work. JADE platform (Bellifemine et al., 2007) provides an infrastructure for multi-agent systems development. It maintains ready-to-use agents communications through message passing. As part of our work in norm identification requires agents to communicate with other agents, JADE is a suitable choice for that.

In the following section we present our view of norm. In Section 2.2 we present an overview of BDI agent. JADE technology is presented in Section 2.3. In Section 2.4 we present a brief introduction of Event Calculus.

## 2.1 What is a norm?

The concept of norm has been used by previous researchers to denote a way to coordinate, regulate, control and predict agent behaviour in open multi-agent systems (Shoham and Tennenholtz, 1995; Verhagen, 2000; Boella et al., 2009). Due to the use of norm in different disciplines, there are several conceptions of norm.

According to López (2003), "*norms prescribe how agents ought to behave in specific situations, and they can make the performance of a system more effective by constraining the behaviour of its components*". So this definition sees norm as a behaviour description. While Savarimuthu and Cranefield (2009) see norms as behavioural expectations: "*norms are expectations of an agent about the behaviour of other agents in the society*". Ostrom (2014) defines norms as "*shared understandings about actions that are obligatory, permitted, or forbidden*". Ostrom sheds the light on the fact that norms should be shared among agents. According to Boella et al. (2006) norm is a "*principle of right action binding upon the members of a group and*

*serving to guide, control, or regulate proper and acceptable behavior*". Regardless of various definitions of norms, norms have been used as behaviour constraints to regulate and coordinate heterogeneous agents' behaviour as well as to foster cooperation and minimize conflicts among agents (Boella and Van Der Torre, 2007a).

Norms describe the standard behaviour of agents, i.e., what an agent should not do (prohibition), what it has to do (obligation) and what it is allowed to but does not have to do (permission). Norms are often described using deontic logic (von Wright, 1968), which studies the relationship between prohibitions, obligations and permissions, as well as norm violation and fulfillment (Boella et al., 2006). The idea of normative multi-agent systems revolves around maintaining a global desirable behaviour of multi-agent systems while preserving agents' autonomy. A normative multi-agent system has thus been defined as:

> A multi-agent system organized by means of mechanisms to represent, communicate, distribute, detect, create, modify, and enforce norms, and mechanisms to deliberate about norms and detect norm violation and fulfillment (Boella et al., 2008).

Norms have been integrated into multi-agent systems to describe standard ideal behaviour and to enforce agents' coordination in environments where a centralized mechanism to enforce behaviour is not available. Researchers have studied methods of using norms to regulate and predict agents' behaviours as well as methods of enforcing agents to comply with the norms without limiting agents' autonomy (Dignum, 1999; Meneguzzi and Luck, 2009; Panagiotidi and Vázquez-Salceda, 2012). For this purpose, two methods have been used, *regimentation* and *enforcement* (Grossi et al., 2007). Agents in the regimentation approach must comply with and can not vio-

late norms (Esteva et al., 2001). This approach drastically limits agents' autonomy and decreases the richness of behaviour in normative multi-agent systems. In the enforcement approach, which this thesis adopts, agents have the choice to respect or violate norms. Basically, their choices have consequences, which are determined by the enforcement regime. A sanction might be applied to agents who violate a norm as a means of enforcement. Issuing sanctions to violator agents is an incentive for agents to comply with norms. The objective of such enforcement is to produce stable and predictable behaviour in the overall system (Castelfranchi, 2003).

Researchers have studied different research lines in normative multi-agent systems, such as the emergence of norms, norm enforcement and norm adoption. Some researchers assume that agents already know the norms of their societies at design time (Conte and Castelfranchi, 1995). Others assume that norms are assigned by a leader or a legislator (Boman, 1999). Assuming that the norms of a society are subject to change or even disappear, agents need a mechanism to infer these changes. Similarly, when an agent joins a new society, it requires a mechanism to infer the norms of the joined society.

### 2.1.1 Our view of norms

In this thesis we study three types of norms: permission, prohibition and obligation. Our definition of norms follows Anderson's reductionist view (Anderson, 1958), which states that norm violation is necessarily followed by a sanction (Soeteman, 2001). This view is rational because if there is no punishment for norm violation, then agents do not adhere to norms, which in turn decreases the importance of the norm concept. Punishment may have several forms, such as paying a fine, reducing privileges, feeling

shame, expulsion from society, etc. Informally, we define norms as follows (for a formal representation see Chapter 5):

- **Prohibition norm**

  In a particular context, if the occurrence of a sequence of actions (or world state) is subject to punishment, then this sequence of actions (or world state) is prohibited in that context.

- **Obligation norm**

  If the nonoccurrence of a prescribed sequence of actions (or world state) in a particular context is followed by punishment, then this sequence of actions (or world state) is obligated in that context. The fulfillment of this sequence of actions (or world state) might also be subject to a reward.

- **Permission norm**

  In a particular context, if the occurrence of a sequence of actions (or world state) is not subject to punishment, then this sequence of actions (or world state) is permitted in that context.

In this thesis, we assume the following properties for norms:

1. Norms are soft constraints on actions or states of affairs; in order to preserve agents' autonomy, agents have the choice respecting or violating norms.

2. The violation of a norm may be subject to punishment, while compliance with a norm may be subject to reward.

3. Norms are shared among the individuals of a society; they are a kind of mutual agreement among these individuals with respect to how members of the society

should behave.

4. Norms are subject to change and can vanish altogether. Norms are not fixed; it is possible for norms to lose their importance and disappear from a society, such that actions that were once prohibited become permitted. There is also the possibility for new norms to emerge.

Norm in our view is composed of the following parts:

- `D` is the deontic type which refers to obliged, prohibited or permitted behaviour.
- `C` is the optional norm's context. The specified sequence of actions is obliged, prohibited or permitted if `C` is a logical consequence of the agent's belief base. When `C` is absent, it means that the norm is applicable under any circumstance. `C` is composed of two possible components $\beta$ and $\alpha$; $\beta$ consists of `holdsAt` (see Section 2.4) predicates that describe a particular world state, while $\alpha$ is an EC formula that represents a sequence of actions.
- `Seq` is a sequence of action(s) that agents are not supposed to perform, have to perform or may perform in case of prohibition, obligation, or permission respectively. Note that `Seq` is different than $\alpha$, since $\alpha$ is part of the context condition (actions that trigger norm activation) whereas `Seq` is the object of the norm's deontic type (actions that are forbidden, obliged or permitted).
- `S` is the sanction that will be applied if the norm has been violated or unfulfilled.
- `R` is the reward that agents may get if they fulfill an obligation norm.

Punishment and rewards may have different forms such as feelings of shame or anxiety, greeting, respect and reputation. In this work we simplify the view of `S` and

`R` to a scalar integer number.

## 2.2 BDI agents

Beliefs, Desires and Intentions (BDI; Bratman, 1987) is one of the most widely studied architectures to implement practical reasoning in multi-agent systems. The BDI architecture came as a balancing reactive and deliberative architecture for goal-oriented agents (Bratman et al., 1988). The BDI architecture is also widely used in the definition of agent programming languages, such as the AgentSpeak(L) programming language (Rao, 1996), arguably the most widely studied of this kind of programming language.

According to Bratman (1987), a folk-psychological theory of human agency and decision-making is centered around three mental components: beliefs, desires and intentions. For example, this morning I have a desire to attend a public talk at Memorial University. There are three possible plans to go from my home to the university: taking the bus, biking, or walking. I believe that today is a nice and sunny day. I also believe that walking is a very healthy habit. When I choose and commit to walk, then the plan of walking to the university becomes my intention. Hence, the result of my mental states, belief, desire and intention, is the plan of walking from home to the university.

Similarly, agents can be designed to include beliefs, desires and intentions (or BDIs) and their behaviour will also be affected by these mental states (Cohen and Levesque, 1990; Rao and Georgeff, 1995). In this case, *beliefs* describe information about the environment. Agents capture their belief through perceiving the environ-

ment and through communications with other agents. An agent's beliefs are evaluated from the agent's point of view; an agent's belief is changeable and it is not necessary to be a knowledge about the environment. *Desires* are the states of affairs an agent wants to bring about. Agents may have several desires, which may be conflict desires, so they need to deliberate to decide which desires to achieve. When agents commit to pursue one desire it becomes the agent's *intention*. Bratman et al. (1988) differentiate present-directed intention and future-directed intention. The former directly produces behaviour, whereas the later makes the agent planning for a particular behaviour and adopt or drop other intentions. Most works on BDI-agent concerned with the later sense of intention where the committed plans to fulfill desires represent the intentions (Cohen and Levesque, 1990).

The BDI agent architecture is based on a practical reasoning process which includes two stages, deliberation and means-end reasoning. During deliberation, agents determine the states of affairs they want to achieve. Then, in the means-ends reasoning stage, agents determine how they will achieve the determined states of affairs. The results of the latter stage are then implemented by plans (Wooldridge, 2002). For example, after a process of deliberation, I decided to achieve the desire of "going to the university". Then, during the means-ends reasoning stage, I chose the plan (taking the bus, biking, or walking) that best allowed me to fulfill my desire.

After Bratman introduced the theory of BDI to describe human behaviours, Rao and Georgeff (1995) adopted the theory for a more formal model suitable to multi-agent systems research. They designed a practical reasoning architecture based on the BDI concept. This architecture is called the Procedural Reasoning System (PRS). The PRS architecture is shown in Figure 2.1; the Figure illustrates the four main components of the PRS: beliefs, desires, intentions and plan library. A plan

specifies a sequence of actions or sub-goals for achieving a particular desire. These four components are controlled by the *interpreter* which is responsible about belief update, generating new desires, choosing from those desires some intentions and selecting actions to be performed on the basis of current intention.

Most historical and current BDI systems are based on PRS (Braubach et al., 2003; Bellifemine et al., 2007). PRS reduces the abstract schema of desires and intentions to a more concrete concept of goals and plans. While making slight modifications to the original concepts of mental states, the most significant variation is defining goals as a consistent set of desires that can be achieved simultaneously. As a result of this modification, the effort of complex goal deliberation is avoided.

In spite of the dominant interest in PRS and BDI agents, there were no precise or specific description of their behaviour. That leads to various implementations for the BDI architecture, such as dMARS (d'Inverno et al., 1998), Jadex (Braubach et al., 2003), AgentSpeak (Rao, 1996) and Jason (Bordini and Hübner, 2006). Currently, Jason is the most active and dominant implementation of BDI architecture; next we give an overview of Jason.

### 2.2.1 Jason

Jason (Bordini and Hübner, 2006) is a Java-based interpreter for an extended version of AgentSpeak(L) (d'Inverno et al., 1998). Agents in Jason use belief-bases that express information about the environment in which agents are situated. They also use logic programming constructs that, unlike traditional AgentSpeak(L), allow Prolog-like logical rules in the agent definition.

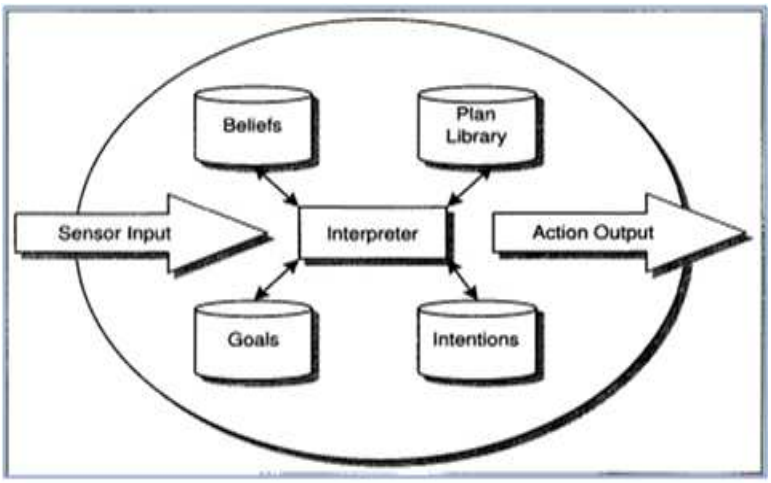


Figure 2.1: BDI Agent architecture PRS (taken from d'Inverno et al., 1998) with permission from the publisher

Since we use the AgentSpeak(L) notation throughout this thesis, we briefly review the syntax of Jason. The most basic syntax element in Jason is a predicate, which is represented by alphanumeric strings starting with a lower case character. A predicate represents a fact about the world and may be evaluated as either true or false. Predicates with arity greater than zero have a number of terms. Terms are similar to predicates but they represent objects in the domain and can be either functions (terms with arity greater than zero), constants (representing specific objects in the domain) or variables. Variable notation follows the Prolog standard and starts with an upper case letter or the underscore sign, representing an unnamed variable.

The basic components of a Jason agent are Beliefs, Goals and Plans. Beliefs represent the agent's knowledge about the environment, other agents or the agent itself. A Jason agent stores the properties of the environment that it believes to be true. Each Jason agent has a belief-base in the form of a collection of literals, as in traditional logic programming. The information in the belief-base is represented as a predicate, for example `clear(table)`, which expresses a particular property of

an environment, in this case, nothing on the table. The "+" and "-" symbols are used to represent changes in the belief-base, denoting belief addition and deletion respectively.

The second basic component, known as Goals, expresses properties of the environment that the agent wishes to achieve. Goals are the initiatives that make the agent operate and try to change the environment to a state in which the agent believes the goals are true. There are two types of goals defined in AgentSpeak(L): achievement goals, which are represented by literals prefixed with "!", and test goals, which are represented by literals prefixed by "?".

Normally, Jason agents have at least one plan to fulfill each goal. Each plan has three parts: triggering-event, context and body. The events that may initiate the execution of a plan are called triggering-events. Triggering-events could be addition (+) or deletion (-) of goals or beliefs. Goals and beliefs updates serve as triggers to the execution of hierarchical plans contained in a plan library. In a plan, the triggering-event part is separated from the context part by the symbol ":". The symbol "<-" separates the context and the body. In rules, the symbol ":-" separates a rule's left- and right-hand sides. The symbols "&" and "|" indicate a conjunction and a disjunction operator respectively. The syntax for a plan in Jason looks like the following

```
Triggering-event:  Context <- body.
```

Consider the following Jason plan:

```
+clear(Block1):  clear(Block2) <- !on(Block1,Block2).
```

This plan states that if the agent perceives that Block1 becomes clear, and if Block2 is clear, then it intends to achieve the goal of putting Block1 on Block2.

Jason agent architecture and its reasoning cycle illustrated in Figure 2.2. For more details on how Jason interpreter functions, we refer the reader to Bordini et al. (2007). When an event occurs, it is unified with each plan's triggering-event in the plan library. The plans that have unified triggering-event are called relevant plans. The context part of each of the relevant plans is checked against the agent belief base. The relevant plans whose contexts are logical consequences of the current belief base are called applicable plans. The body of a plan is formed of actions, sub-goals or internal action. Example of internal action is `.send(-,-,-)`, which is used for agents communication. A plan is selected from the applicable plans and added to the agent intentions set.

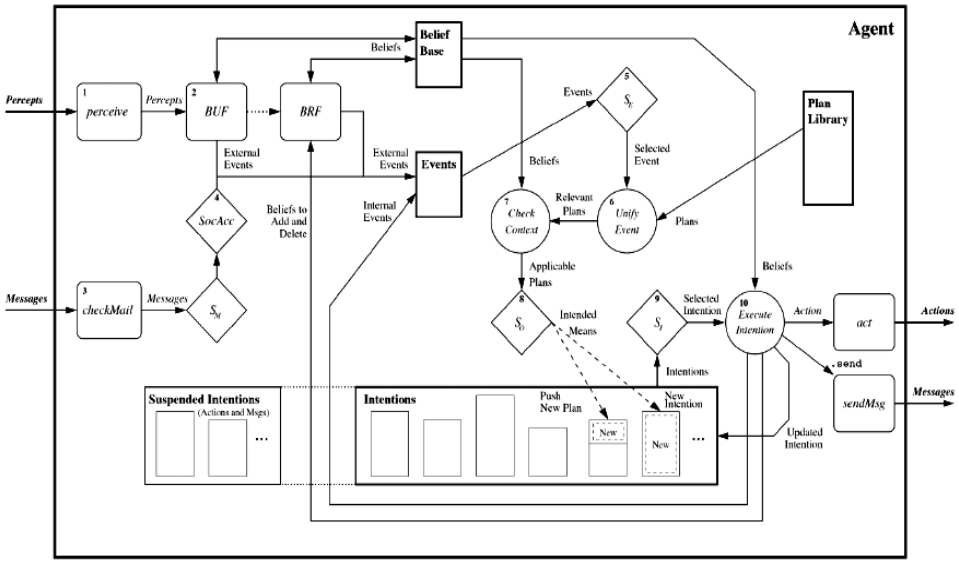


Figure 2.2: The Jason reasoning cycle (taken from Bordini et al., 2007) with permission from the publisher

**Example 2.2.1.** As a simple example using Jason, consider an automated parking

lot which has three agents: `agent1` receives orders from customers and assigns goals to be achieved by `agent2` and `agent3`. `Agent2` is responsible for taking cars from the entrance of the parking lot and parking them inside the lot, while `agent3` is responsible for taking cars out of the parking lot. To simplify the example explanation, suppose that the capacity of the parking lot is three. The Jason code for the three agents is given below (note that we have annotated each plan with a label so that we can refer to them in the text below).

**agent1**

Beliefs

...

Plans

```
+parkIn(X)<-.send(agent2,achieve,park(X)).          (p1)
```

```
+parkOut(X)<-.send(agent3,achieve,leave(X)).        (p2)
```

Plan p1 tells `agent2` to park car `X` in the parking lot. Plan p2 tells `agent3` to take car `X` out of the parking lot. Those plans are triggered when `agent1` receives or perceives a request from a customer to park or retrieve a car. By receiving the request, the predicate `parkIn(X)` or `parkOut(X)` is added to agent's belief-base. The body of each plan in `agent1` above is a message of three arguments: the message receiver (`agent2` in case of plan p1), the type of the message ("achieve" in case of plan p1 and p2, which means that a goal is sent to the receiver), and the goal that the sender wishes to achieve (`!park(X)` and `!leave(X)` in case of plans p1 and p2 respectively).

**agent2**

Beliefs

```
empty(1).
empty(2).
empty(3).
```

Plans

```
+!park(X) : not full <- ?empty(Y);!driveTo(X,Y);
    .send(agent3,tell,parkingAt(X,Y));                    (p1)
    .send(agent3,tell,weHave(X)); -empty(Y).

-!park(X)<- +full;.print("sorry the parking is full").    (p2)

+!driveTo(X,Y)<- .print("the car",X,"is parking at", Y).  (p3)

-!driveTo(X,Y)<-.print("Error, please contact the operator").  (p4)
```

As the belief-base of `agent2` expresses, three spaces are available for parking. Once `agent2` receives an order from `agent1` to achieve a goal (let us say `!park(car1)`), the predicate `park(car1)` is unified with the triggering part of plan p1 and `car1` is unified with `X`. Based on the context of plan p1, the plan is only executed if the parking lot is not full. The purpose of the test goal `?empty(Y)` is to find an empty slot which will be unified with the variable `Y`. After this, the sub-goal `!driveTo(car1,1)` is added to be executed. Plan p3 is triggered when the sub-goal is achieved and `agent2` sends a message to `agent3` telling it `car1` has been parked at slot 1. As a result `agent3` will add the predicate `parkingAt(car1,1)` to its belief-

base. The last step of plan p1 is `-empty(Y)`, which will delete the predicate `empty(1)` from the belief-base of `agent2`. Any action failure in the plan causes the whole plan to fail. Plan p2 is called a failure plan and is triggered only if plan p1 fails. Plan P4 is triggered only if plan p3 fails.

**agent3**

Beliefs

parkingAt(car1,1).

Plans

```
+!leave(X) : true <- ?parkingAt(X,Y); !driveOut(X);        (p1)
    +empty(Y); .send(agent1,tell,empty(Y)).

-!leave(X): not parkingAt(X,Y)<-                           (p2)
   .print("sorry car ",X," does not exist").

+!driveOut(X) <- !driveTo(X,out).                          (p3)

+!driveTo(X,out)<-.print("The car ",X, " is out").         (p4)

-!driveTo(X,out)<-.print("please contact the operator").   (p5).
```

When `agent3` receives a message from `agent1` asking it to retrieve `car1` from parking (`!leave(car1)`), plan p1 is triggered. Using the test goal `?parkingAt(X,Y)`, the parking slot of `car1` is determined. If the test goal `?parkingAt(X,Y)` fails, `agent3` looks for an implemented plan for `?parkingAt(X,Y)`. If there is no such plan, then plan p1 fails. By achieving the goal `+!driveTo(X,out)`, car `X` is going to be driven

to the position `out`. Plan P5 is a failure plan it is called only when plan p4 fails.

## 2.3 JADE

JADE (Java Agent DEvelopment framework) (Bellifemine et al., 2007) is a software platform that provides basic infrastructure for building multi-agent systems. It is one of the most popular agent-oriented middleware programs in use today. Building multi-agent systems on top of JADE, which provides the domain-independent infrastructure, helps the developers to focus on the business logic of their systems instead of dealing with the specification of agents' communications and interoperability. Agents' communications on JADE platform comply with the FIPA specifications.

JADE platform is composed of agent containers that can be distributed over the network. A container is a Java process which provides the needed services for hosting and executing agents. One special container called main container forms the starting point of the platform; all other containers need to register with the main container. The main container manages the container table (CT) which registers the addresses of all other containers in the platform.

Each container manages the global agent descriptor table (GADT) which registers all agents that exist in the platform including their current state and location. Each container maintains a local agent descriptor table (LADT), which is a cash of GADT. When an agent wants to message another agent, it looks for the receiver address in LADT. If it could not find it then it looks in GADT. Figure 2.3 illustrates the main elements of JADE platform.

The main container host two special agents: agent management system (AMS) and directory facilitator (DF), which are automatically started once JADE is launched. The roles of these agents are specified by the FIPA agent management system standards (FIPA, 2004). Every agent in the system should register with the AMS agent. AMS supervises the whole platform. It forms the contact point for all agents' communications in addition to managing agents' life cycle. All JADE agents need the DF agent to register their services or to look for other agents' services.

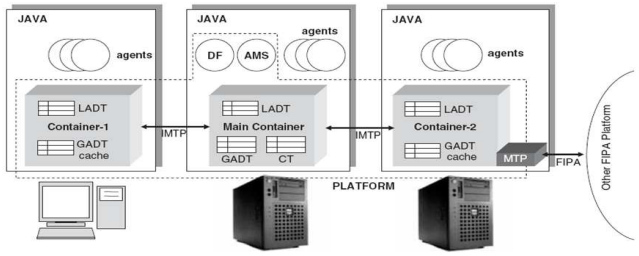


Figure 2.3: JADE platform architecture (taken from Bellifemine et al., 2007) with permission from the publisher

## 2.4 Event calculus

Event calculus (EC) is a logical framework which uses predicates and axioms to represent and reason about actions and their effects. EC was originally proposed in logic programming (Kowalski and Sergot, 1989) with the purpose of affirming that as a result of executing an action, a particular property is initialized to be true at a specific time-point and that no action occurs later to terminate this property.

EC can be used as a foundation for different reasoning tasks like deductive, abductive and inductive reasoning (Shanahan, 1999). In case of deductive reasoning, the narrative events (*which events happened and when*) and the effects of actions

(*what actions do*) are given, while the result (*what is true*) is sought. Figure 2.4 depicts how EC operates. In abductive reasoning, the sequence of actions that lead some properties to be true are sought. Referring to Figure 2.4, (*what actions do*) and (*what is true when*), are given in the case of abductive reasoning and (*what happens*) is required. In the case of inductive reasoning, (*what happens when*) and (*what is true when*) are given, and (*what actions do*) is required.

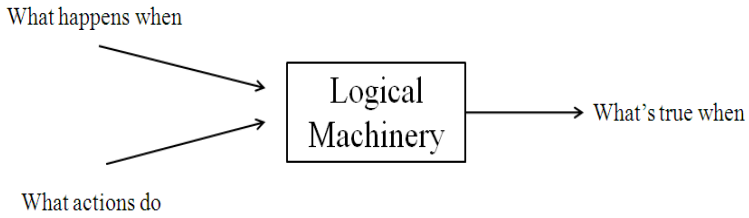


Figure 2.4: How event calculus functions (taken from Shanahan, 1999) with permission from the publisher

The basic idea of EC is that a particular fluent is turned to be true at a particular time-point as a result of performing a particular action or a sequence of actions, and in the meantime no action occurred to terminate that fluent. The basic ingredients of EC are Actions `A`, Fluents `F`, Time-points `T`, a set of domain independent axioms and a set of domain dependent axioms that describe the desired model.

A fluent is a property whose values are subject to change at different points in time. Fluents in EC are reified, which means they are not formalized by predicates but by functions. One of the basic EC predicates is `HoldsAt`, which takes two arguments: Fluent and Time-point. Consider the predicate `HoldsAt(on(book,table),t)` which means that the book is on the table at time `t`. Note that `HoldsAt` is a predicate and `on` is a function.

Referring to Figure 2.4, usually the narrative part (*What happens when*) is represented by the predicate `happens` which takes two arguments: Action and Time-point. The effect of the actions (*What actions do*) is represented by the predicates `initiates` and `terminates`, both of which take three arguments: Action, Fluent and Time-point. The set of domain independent axioms below and the set of predicates shown in Table 2.1 represent the language of basic EC.

**EC1**: `clipped(T1,F,T2)`$\leftarrow$

$\exists$ `A,T[happens(A,T)` & `T1`$\leq$`T` & `T<` `T2` & `terminates(A,F,T)`

This states that fluent `F` is terminated by the occurrence of action `A` between time `T1` and `T2`.

**EC2:** `declipped(T1,F,T2)`$\leftarrow$

$\exists$ `A,T[happens(A,T)` & `T1`$\leq$`T` & `T<T2` & `initiates(A,F,T)`

This states that fluent `F` is initiated by the occurrence of action `A` between time `T1` and `T2`.

**EC3:** `holdsAt(F,T2)`$\leftarrow$ `happens(A,T1)` & `initiates(A,F,T1)` &
`T1<T2` & $\neg$`clipped(T1,F,T2)`

This states that fluent `F` holds at time `T2` if action `A` occurs at time `T1` and `F` has not terminated between `T2` and `T1`.

**EC4:** $\neg$`holdsAt(F,T2)`$\leftarrow$ `happens(A,T1)` & `terminates(A,F,T1)` &
`T1<T2` & $\neg$`declipped(T1,F,T2)`

This states that fluent `F` does not hold at time `T2`, while action `A` terminates fluent `F` at time `T1` and no action occurred between time `T1` and `T2` to re-initiate fluent `F`.

**EC5:** `holdsAt(F,T2)`$\leftarrow$ `holdsAt(F,T1)` & `T1<T2` & $\neg$`clipped(T1,F,T2)`

**EC6:** $\neg$`holdsAt(F,T2)`$\leftarrow$ $\neg$ `holdsAt(F,T1)` & `T1<T2` & $\neg$`declipped(T1,F,T2)`

EC5 and EC6 axiomatize the persistence of fluents; that is, they state that the value of fluents are changed only via the occurrence of actions that may initiate or terminate fluents.

The definition of the predicates `happens`, `initiates` and `terminates` are defined according to the domain we deal with. Consider the following example taken from (Miller and Shanahan, 1999).

**Example 2.4.1.** A robot can go inside and outside a room through a door. The door can be locked and unlocked using an electric key. In this scenario we have three fluents and three actions. The fluents are `Inside` (the robot exists inside the room), `HasKey` (the robot is holding the key) and `Locked` (the door is locked). The actions are `Insert` (insert the key in the door); `GoThrough` (the robot passes though the door); and `Pickup` (the robot picks up the key). The occurrence of `Insert` makes the door locked if it is unlocked or unlocked if it is locked. If `GoThrough` is performed when the robot is inside the room then the robot's status becomes outside the room and vice versa. When the robot performs `Pickup`, the robot picks up the key and the fluent `HasKey` becomes true.

Consider the following domain-dependent axioms for the example above:

**R1:** `initiates(Pickup,HasKey,T)`

**R2:** `initiates(Insert,Locked,T)`$\leftarrow$

$\neg$`holdsAt(Locked,T)` & `holdsAt(HasKey,T)`

**R3:** `initiates(GoThrough,Inside,T)`$\leftarrow$

$\neg$`holdsAt(Locked,T)` & $\neg$`holdsAt(Inside,T)`

**R4:** `initiates(GoThrough,Inside,T)`

$\neg$`holdsAt(Locked,T)` & `holdsAt(Inside,T)`

**R5:** `initiates(Insert,Locked,T)`$\leftarrow$

`holdsAt(Locked,T)` & `holdsAt(HasKey,T)`

Initially, the door is closed and the robot is inside the room at time 0 and the following actions have been performed:

- `happens(Pickup,2)`
- `happens(Insert,4)`
- `happens(GoThrough,6)`

Using the domain independent axioms (EC1,...,EC6), the domain dependent axioms (R1,...,R5) and the sequence of actions that occurred, we can affirm that the robot is no longer inside the room at time 8, i.e., $\neg$`holdsAt(Inside,8)`, see Figure 2.5 which illustrates this scenario.

EC is distinguished by its simplicity in describing concepts, and it easily allows

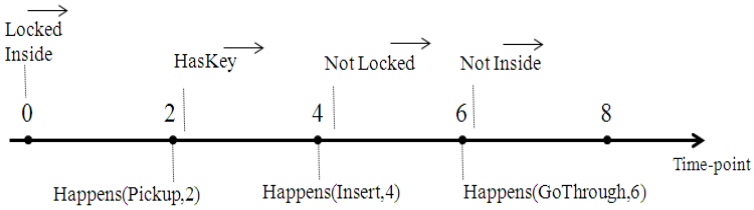


Figure 2.5: The flow of the robot's actions and their effects

Table 2.1: The predicates of event calculus

| Predicate | Meaning |
|---|---|
| happens(A,T) | Action A occurs at time T |
| holdsAt(F,T) | Fluent F is true at time T |
| terminate(A,F,T) | Occurrence of action A at time T will make fluent F false after time T |
| initiates(A,F,T) | Occurrence of action A at time T will make fluent F true after time T |
| initiallyp(F) | Fluent F holds from time 0 |
| clipped(T,F,Tn) | Fluent F is terminated between time T and Tn |
| $<$, $>$, $\leq$, $\geq$ | Standard order relation for time |

the implementation of concept specifications since it is described in a logic programming format. Therefore, EC has been used for representing different concepts in multi-agent systems (see Artikis et al., 2005; Fornara and Colombetti, 2009).

# Chapter 3

# Identifying Norms in Open Multi-agent Societies

From the perspective of individual agents, it is important to know which behaviours are acceptable or unacceptable in a particular society. Identifying norms helps agents to adapt their plans for achieving their goals (Andrighetto et al., 2008).

When a new agent joins a software agent society, it needs to figure out the norms applied in this society to avoid potential punishments that may result from norms violation. One way to address this problem is by inferring prohibition norm based on observing sanction events (Savarimuthu et al., 2011). Under this approach, there are two possibilities: (1) the newcomer agent can act immediately, without knowing the prohibition norms, and can be sanctioned as a result; (2) the agent can wait to observe other agents being sanctioned due to the violation of prohibition norms, and then infer the prohibition norms in the process. As a result the newcomer agent could lose resources by acting recklessly without knowing the norms and hence be

sanctioned, or waste time by waiting for the occurrence of sanction events.

By utilizing permission norms, both of these possibilities can be avoided. Our recognizer agent (the newcomer) does not have to wait for observing sanction events in order to act in the joined society, but it can infer permission norms by observing regular events. As a result, the newcomer agent operates in the society in due course.

As new norms can emerge and known norms can disappear, norm identification algorithms should be able to identify these changes. Savarimuthu (2011) is one of the first researchers to address this problem by observing special sanction events occurring among agents to recognize prohibition norms. According to Savarimuthu's approach, if a particular identified prohibition norm is not detected by his algorithm for a particular time, then this norm is revoked from the agent's belief base. We argue that this approach can mistakenly revoke prohibition norms if those norms have not been violated for a particular time. For example, using a traffic system analogy, an agent A removes the prohibition norm of running red lights if it does not observe other agents running the red lights for a particular time. As an alternative approach, we utilize permission norms for detecting the repealed norms and we compare the two approaches.

Accordingly, we introduce the idea of recognizing permission norms inferred by observing regular (non-sanction) events. We present the drawbacks of the known Verification Component (VC), which is part of the norm identification approach presented by Savarimuthu et al. (2010b), and present a modification that makes agents able to dynamically identify norm changes that occur in a society and function in open multi-agent societies.

To summarize, identifying permission norms for multi-agent systems is significant

at least for two reasons. It helps agents to detect the changes that may occur for prohibition norms (i.e., agents detect that a particular prohibition norm is no longer prohibited when it is identified as permitted), and it helps agents to prefer actions that are known as permitted over actions that are not known to be either permitted or prohibited.

In this chapter we propose an agent architecture and algorithms for inferring prohibition and permission norms. Our proposed architecture identifies and recognizes changes in norms that can occur or evolve in multi-agent systems. This architecture is based on observing both sanctioned and non-sanctioned events. Using JADE (see Section 2.3), the software agent development environment, we construct a restaurant interaction scenario to test our architecture and algorithms. We use this scenario to demonstrate how dynamic permission and prohibition norms can be identified. Regular actions are basic actions performed to achieve a particular goal. Usually regular actions are defined at design time. Hence, each agent knows what actions it can perform beforehand. A sanction action represents a penalty an agent receives for violating a norm. A police agent play the role of social authority by monitoring violations and applying sanctions.

This chapter is structured as follows. In Section 3.1, an overview of the proposed architecture is presented. The algorithms for norm identification are presented in Section 3.2. A proposed modification on the VC is given in Section 3.3. In Section 3.4, we demonstrate the limitations of the VC used by Savarimuthu et al. (2010b) and address these limitations by proposing a modified verification component (MVC). We also extend the dynamic verification component (DVC) presented in Section 3.3 by proposing the modified dynamic verification component (MDVC), which allows an agent to infer newly adopted and dropped norms in open multi-agent societies

(Section 3.5). We show that MDVC preserves the consistency among norms before adopting an identified norm. We summarize the chapter in Section 3.6.

## 3.1 Overview of the architecture of norm identification

The purpose of the proposed architecture and algorithms is to recognize and dynamically infer two types of norms in multi-agent systems: permission and prohibition.

We assume that our agent is able to observe the communication and the interactions that occur among the individuals of a society. We also assume that our agent is able to distinguish between regular and sanction events. The architecture of our agent maintains a queue for each agent in its vicinity and stores other agents' activities in their corresponding queue. The actions are stored in the queue from right to left, with the rightmost action being the recently added. Agents are considered to be in our agent's vicinity if their performed events are observable by our agent (the recognizer agent).

Based on the observed data, the recognizer agent applies two algorithms, one for identifying permission norms and the other for identifying prohibition norms. Our proposed architecture maintains consistency among permission and prohibition norms. Therefore, the recognizer agent does not have a sequence of events which is identified as prohibited and permitted at the same time.

Our proposed architecture for norm identification is composed of storage components (depicted by cylinder shapes) and processing components (depicted by rect-

angular shapes). There are three storage components: event base, prohibition norm base and permission norm base. Figure 3.1 illustrates the architecture components, which are described in detail below.

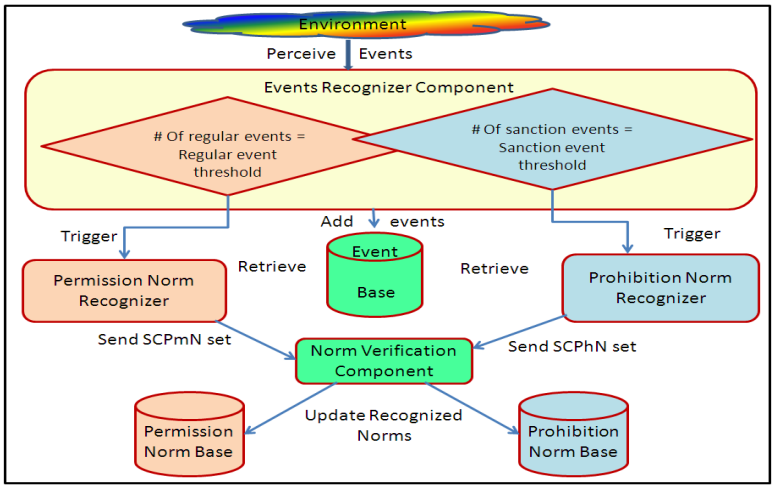


Figure 3.1: Our proposed internal recognizer agent architecture for identifying permission and prohibition norms

**Prohibition Norm Base** (`ProhNrmBase`): stores the prohibition norms of a society. Its contents are acquired by our prohibition norm identification algorithm.

**Permission Norm Base** (`PermNrmBase`): stores the permission norms of a society. As with the prohibition norm base, its contents are acquired by the permission norms identification algorithm.

**Event Base** (`EventBase`): maintains a queue for each observable agent. Any agent's observable events are saved in the corresponding agent's queue in `EventBase`. `EventBase` is composed of queues, where each queue stores the events that are performed by a particular agent. The events are stored in the queue according to the order in which they occur.

There are four processing components: events recognizer, prohibition norm recognizer, permission norm recognizer, and norm verification. These are described in detail below.

**Events Recognizer Component**: observes the events that occur in the society. We assume that this component is able to distinguish between regular and sanction events. The perceived events are saved in the `EventBase`, particularly in the queue that is associated with the agent that performs those events. These data are used for norm identification processes.

**Prohibition Norms Recognizer Component**: is responsible for identifying the set of selected candidate prohibition norms (`SCPhN`) based on event observations. The algorithm for identifying prohibition norms is triggered every time the events recognizer component perceives a number of special events (sanction events) equal to a threshold, `SanThresh`.

**Permission Norms Recognizer component**: finds the set of selected candidate permission norms (`SCPmN`) based on observing regular events. The algorithm for permission norm identification is triggered every time a queue has a number of consecutive regular events equal to a threshold, `RegThresh`.

**Norms Verification Component** (VC) (Savarimuthu et al., 2011): is applied to each candidate norm. In the agent performing norm identification, the VC selects other agents and asks them whether the selected candidate norm is an actual norm.

## 3.2 Norm Identification Algorithms

In this section we present algorithms for norm identification and describe how these algorithms work with the architecture components discussed in Section 3.1.

### 3.2.1 Event recognizer

Our recognizer agent continuously perceives other agents' events and stores them in the queues that are associated with the agents that performed the events. These queues form `EventBase` of the recognizer agent. The actions in each queue represent a sequence of actions. Algorithm 1 determines when the norm identification algorithms are triggered. Every time the `SancNum` (the number of observed sanction events) is greater than or equal to `SanThresh`, the prohibition norm recognizer algorithm is triggered (Lines 5 and 6), where `SanThresh` represents the number of sanction events required to trigger the prohibition norm recognizer algorithm. The permission norm recognizer algorithm is triggered when `RegNum` (the number of consecutive regular events)in a particular queue equals `RegThresh`, (Lines 8 and 9) while no sanction event occurs in that queue. `RegThresh` represents the number of observed consecutive regular events required to call the permission norm recognizer algorithm. In our experiments `RegThresh` is equal to `QueueSize`, where `QueueSize` represents the number of observable events that the recognizer agent can hold for an observed agent.

**Algorithm 1** Main algorithm

```
1: function EventsRecognizer
2:     while true do
3:         Perceive events α
4:         Add α to the corresponding agents' queue in the EventBase
5:         if SancNum ≥ SanThresh then
6:             ProhNrmIdent(EventBase)  /*Algorithm 2*/
7:             SancNum ← 0
8:         end if
9:         if SancNum = 0 & RegNum ≥ RegThresh then
10:            PermNrmIdent(EventBase)  /*Algorithm 3*/
11:            RegNum ← 0
12:        end if
13:    end while
14: end function
```

### 3.2.2 Prohibition norm recognizer

The purpose of this algorithm is to construct a set of candidate prohibition norms by processing `EventBase`. We define a *collection* as a group of objects that might be unordered or duplicated. We define candidate prohibition norms collection (`CPhNC`) as a collection of sequences of actions. Out of the constructed `CPhNC`, a set of candidate norms is selected, called the selected candidate prohibition norms (`SCPhN`) set. The candidate norms with a frequency greater than a particular threshold (`SelectThresh`) are selected from `CPhNC` and stored in `SCPhN`. To explain how our agent infers prohibition norms, consider the example given below.

Let the recognizer agent have two agents, `A` and `B`, in its vicinity, where agents are in the vicinity of the recognizer agent if it can observe their actions. Let the queue size be equal to four (i.e., it can save up to four actions). Assume that `SanThresh` and `SelectThresh` equal one. Suppose that the sequences of actions that are performed by agents `A` and `B`, which are stored in their corresponding queues in the recognizer

agent's architecture, are as follows:

Agent `A`'s queue = **(E2, E2, E3, S)**.

Agent `B`'s queue = **(E3, E2, S, E1)**.

This means that our agent made the following observations: after agent `A` performed action **E2**, **E2** and **E3** it received a sanction **S**, where **E** represents a regular event and **S** represents a sanction event. The prohibition norm recognizer algorithm (Algorithm 2) aims to find the sequence of actions (prohibition norm) that might be the reason behind issuing the sanction event **S**. The process in this algorithm is composed of three steps:

**Step 1**: for each queue, all the sub-sequences of the sequence of regular actions are formed as explained below (see Algorithm 2, Line 5), where `findSubSeq()` is a function that returns all the sub-sequences of a giving sequence.

(a) The sub-sequences events that result from agent `A`'s queue (Line 5) are stored in `CPhNC`:

`CPhNC` = [**(E2)**, **(E2)**, **(E3)**, **(E2,E2)**, **(E2,E3)**, **(E2,E3)**, **(E2,E2,E3)**]

(b) The sequence of events that causes the sanction event (the prohibition norm) should be one of the sub-sequences above. In Line 6, the repeated sequences are eliminated from `CPhNC` using the `remRepeated(Col)` function that removes the duplication from collection `Col`. Hence the contents of `CPhNC` become

`CPhNC` = [**(E2)**, **(E3)**, **(E2,E2)**, **(E2,E3)**, **(E2,E2,E3)**]

(c) Similarly, the result of finding all the sub-sequences of the sequence events performed by agent `B` are:

**[(E3), (E2), (E3,E2)]**

(d) All the sub-sequences that result from each queue are added to `CPhNC`. The candidate prohibition norms collection for the example above is:

`CPhNC` = **[(E2), (E3), (E2,E2), (E2,E3), (E2,E2,E3), (E3), (E2), (E3,E2)]**

**Step 2**: the candidate prohibition norms with a frequency greater than `SelectThresh` (in our experiment, this is a frequency >1) are selected from `CPhNC` and stored in `SCPhN` (Algorithm 2, Line 10). `freq(X,Col)` is the function that returns the number of times object `X` appeared in a collection `Col`. Hence the contents of `SCPhN` become `SCPhN` = **{(E2), (E3)}**

**Step 3**: in Line 11, the `SCPhN` set is passed to the verification component, Algorithm 4, which verifies whether the selected candidate prohibition norms are actual norms in the society.

**Algorithm 2** Prohibition norm identification algorithm

```
1:  function ProhNrmIdent(EventBase)
2:      for all Q ∈ EventBase do
3:          for all S event in Q do
4:              SeqReg ← the sequence of regular events that precede S
5:              Col ← findSubSeq(SeqReg)
6:              Col ← remRepeated(Col)
7:              CPhNC ← CPhNC ∪ Col
8:          end for
9:      end for
10:     SCPhN ← ∀ x ∈ CPhNC where freq(x,CPhNC)>SelectThresh
11:     ProhNrmVerifi(SCPhN) /*Algorithm 4*/
12: end function
```

### 3.2.3 Permission norm recognizer

The permission norm recognizer algorithm recognizes sequences of actions that the agents perform without causing a sanction (permission norms). The idea here is to find patterns of actions that occurs without causing a sanction. A repeated pattern of actions in `EventBase` that occurs without a sanction event gives an indication that the pattern does not cause punishment. Consequently it is identified as a candidate permission norm. We define the candidate permission norm collection (`CPmNC`) as a collection of objects that might be unordered or duplicated.

Permission norm identification algorithm (Algorithm 3) is only triggered when at least one queue has no sanction event and the number of regular events hits the regular events threshold. In this case, even if other queues can have sanction events, Algorithm 3 can use those queues to identify permission norms. If a recognizer agent observes the following actions: E1, S, E2 and E4, it can utilize this data to identify permission norms, because it is possible that E2 and E4 are permitted (as they appear after the sanction event S and they are not followed by another sanction event). Therefore, `SeqReg` that appears in Algorithm 3 contains the sequence of regular events that is not followed by a sanction event.

There are three steps in this component (see Algorithm 3). To illustrate, let us again consider the example given in Section 3.2.2, repeated here for convenience. Let the recognizer agent have two agents, `A` and `B`, in its vicinity. Agents are in the vicinity of the recognizer agent if it can observe their actions. Let the queue size be equal to four (i.e., it can save up to four actions). Assume that `SanThresh` and `SelectThresh` equal one. This time, suppose that the sequences of actions that are performed by agents `A` and `B` are as follows:

Agent A's queue = **(E2,E2,E3,E4)**.

Agent B's queue = **(E3,S,E4,E1)**.

**Step 1**: as in the previous section, for each queue all the sub-sequences of the sequence of regular actions are constructed as explained below (Algorithm 3, Line 4). The result of finding all the sub-sequences of agent A's queue are:

**[(E2), (E2), (E3), (E4), (E2,E2), (E2,E3), (E2,E4), (E2,E3), (E2,E4), (E3,E4), (E2,E2,E3), (E2,E2,E4), (E2,E3,E4), (E2,E3,E4), (E2,E2,E3,E4)]**

In the permission norm identification process, the duplicated sub-sequences are not deleted. No deletion occurs because, in a sequence in which no sanction event occurs, repeated sub-sequences indicate that this repeated sub-sequence might be permitted. The result of finding all sub-sequences of the actions performed by agent B is:

**[(E4), (E1), (E4,E1)]**.

The sub-sequences from the queues of agent A and B that are repeated more than SelectThresh times (which is set to 1 here) are added to the CPmNC. For the example above, we have:

CPmNC = **[(E2), (E2,E3), (E2,E4), (E2,E3,E4)]**.

Nothing is added from agent B's sub-sequences since there are no repeated sub-sequences.

**Step 2**: in Line 6, the duplicated sub-sequences are removed from CPmNC and the resulting version of CPmNC is added to SCPmN set:

SCPmN = **{(E2), (E2,E3), (E2,E4), (E2,E3,E4)}**

**Step 3**: in Line 9, SCPmN is passed to the verification component, which verifies whether the SCPmNs are accepted as actual permission norms.

The function `findSubSeq(SeqReg)` finds all the sub-sequences of a given sequence, which is equal to $2^k - 1$, where $k$ is the length of the given sequence. The time complexity of Algorithms 2 and 3 is $O(n*2^m)$, where $n$ is the number of observed agents and $m$ is the number of consecutive regular actions. However, $m$ is relatively small number, in our experiments $m \leq 8$. This is a reasonable value because the sanction event usually will not be far away from the violated actions.

**Algorithm 3** Permission norm identification algorithm

```
1: function PermNrmIdent(EventBase)
2:     for all Q ∈ EventBase do
3:         SeqReg ← the sequence of regular events in Q that is not followed by a
   sanction event
4:         Col ← findSubSeq(SeqReg)
5:         CPmNC ← ∀ x ∈ Col where freq(x,Col)>SelectThresh
6:         CPmNC ← remRepeated(CPmNC)
7:     end for
8:     SCPmN ← CPmNC
9:     PerNrmVerifi(SCPmN) /*Algorithm 5*/
10: end function
```

### 3.2.4 Norm verification component (VC)

We replicate Savarimuthu et al. (2011)'s VC as a prelude to our contributions. We argue that the original VC is not suitable for open multi-agent systems, in addition to the fact that it does not deal with permission norm (VC for permission norm is our addition). The norm VC is based on asking other agents whether the candidate norm is known as an actual norm. The agents respond based on their experience in the society. To avoid accepting false information, several agents are asked and the majority answer is taken. Figure 3.2 illustrates this process. The `NumofWiseAgents` parameter keeps the number of agents which the recognizer agent will ask. In our

work, the recognizer agent selects the agents that are to be queried randomly.

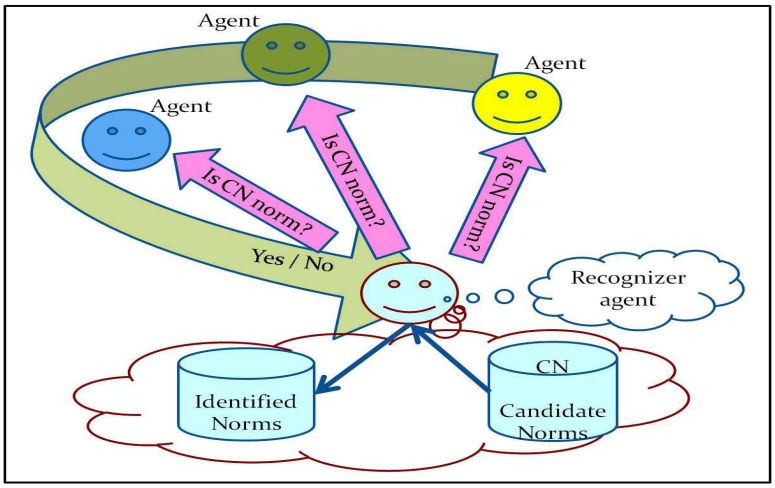


Figure 3.2: The verification component (VC). The recognizer agent picks a candidate norm (CN) from the candidate norms base and asks three agents whether CN is a norm or not. If the majority answer is affirmative, then CN is stored as an identified norm.

There are two execution scenarios based on the type of norms we intend to verify. For prohibition norms, the recognizer agent asks whether a selected candidate prohibition norm is a prohibition norm or not. If the majority of the answers are positive, the selected candidate norm is added to the recognizer agent's prohibition norm base; otherwise, the candidate norm is rejected (see Algorithm 4).

In the case of permission norms, the recognizer agent asks whether a selected candidate permission norm is known as a permission norm or not. If the majority of the answers are positive, then the candidate permission norm is added to the recognizer agent's permission norm base; otherwise, the candidate norm is rejected (see Algorithm 5).

**Algorithm 4** Prohibition norms verification algorithm

```
1: function ProhNrmVerifi(SCPhN)
2:     Select NumofWiseAgents agents
3:     for all α ∈ SCPhN do
4:         Ask NumofWiseAgents agents whether α is a prohibition norm.
5:         if the majority answer is Yes then
6:             ProhNrmBase← ProhNrmBase ∪ α
7:         end if
8:     end for
9: end function
```

**Algorithm 5** Permission norms verification algorithm

```
1: function PerNrmVerifi(SCPmN)
2:     Select NumofWiseAgents agents
3:     for all α ∈ SCPmN do
4:         Ask NumofWiseAgents agents whether α is a permission norm.
5:         if the majority answer is Yes then
6:             PermNrmBase← PermNrmBase ∪ α
7:         end if
8:     end for
9: end function
```

## 3.3 The dynamic verification component (DVC)

Sometimes, new norms emerge or disappear from a society. Therefore, agents need a way not only to discover these potential changes but also to keep the norm base in a consistent state. Hence, we present the dynamic verification component (DVC) which is a modification of the VC.

The DVC allows an agent to infer new emerged and repealed norms online. It is composed of two steps. In the first step, the recognizer agent selects agents randomly and asks them to verify whether the selected candidate norm is an actual norm in the society. In the second step, the recognizer agent decides whether to accept or reject the verified norm. This decision is taken based on the majority answer of the selected

agents, as discussed in Section 3.2.4. The DVC attempts to preserve consistency among norms before adopting an identified norm. There are two main scenarios for the DVC.

First scenario: in the case of verifying a candidate prohibition norm (Algorithm 6), if the majority answer is positive (yes, it is a prohibition norm), then the candidate prohibition norm is compared with the permission norms (Lines 6 - 10). If it does not equal any permission norm, then the candidate prohibition norm is stored in the prohibition norm base (Line 11). If the candidate prohibition norm equals a permission norm, then that permission norm is deleted from the permission norm base before the candidate prohibition norm is added to the prohibition norm base (Line 8).

Second scenario: in the case of verifying a candidate permission norm (Algorithm 7), if the majority answer is positive (yes, it is a permission norm), then the candidate permission norm is compared with the prohibition norms (Lines 6 - 10). If it does not equal any of the prohibition norms, it is stored in the permission norm base (Line 11). If it equals a prohibition norm then that prohibition norm is deleted from the prohibition norm base before the candidate permission norm is added to the permission norm base (Line 8).

Consequently, the recognizer agent maintains the consistency among the permission and prohibition norms, never having a sequence of events defined as prohibition norm and permission norm simultaneously.

**Algorithm 6** Dynamic prohibition norms verification algorithm

```
1: function DynProhNrmVerifi(SCPhN)
2:     Randomly select NumofWiseAgents agents
3:     for all α ∈ SCPhN do
4:         Ask NumofWiseAgents agents whether α is a prohibition norm.
5:         if the majority answer is Yes then
6:             for all Pn ∈ PermNrmBase do /*where Pn is a permission norm*/
7:                 if Pn =α then
8:                     PermNrmBase ← PermNrmBase \ Pn
9:                 end if
10:            end for
11:            ProhNrmBase ← ProhNrmBase ∪ α
12:        end if
13:    end for
14: end function
```

**Algorithm 7** Dynamic permission norms verification algorithm

```
1: function DynPerNrmVerifi(SCPmN)
2:     Randomly select NumofWiseAgents agents
3:     for all α ∈ SCPmN do
4:         Ask NumofWiseAgents agents whether α is a permission norm.
5:         if the majority answer is Yes then
6:             for all Fn ∈ ProhNrmBase do /*where Fn is a prohibition norm*/
7:                 if Fn = α then
8:                     ProhNrmBase ← ProhNrmBase \ Fn
9:                 end if
10:            end for
11:            PermNrmBase ← PermNrmBase ∪ α
12:        end if
13:    end for
14: end function
```

## 3.4 The modified verification component (MVC)

In open multi-agent systems, agents can have different internal architectures. Thus, our agent may communicate with agents that do not necessarily know the concept of norm. Further, because agents in open multi-agent systems are not always owned by the same owner, they may have different goals and objectives. Consequently, agents in open multi-agent systems are more likely to be self-interested and unreliable (Huynh et al., 2006).

The VC mechanism has some drawbacks when it is used in open multi-agent societies, especially those societies that are heterogeneous in the sense that agents can have different architectures, possibly created by different designers (Grossi, 2007; Koeppen and Lopez-Sanchez, 2010). Generally, the limitations fall into two categories: agents with no concept of norm and agents with different and contradicting objectives.

In the first case, there are some agents in open multi-agent systems that do not understand or are not aware of the concept of norm, or they might use a different terminology for norm. This means that using the VC and asking those agents whether a particular candidate norm is actually a norm becomes a futile exercise.

In the second case, open multi-agent systems contain agents that are owned by different stakeholders (see Huynh et al., 2006). Consequently, each agent or group of agents may have its own objectives, possibly in conflict with other agents. Accordingly, we need to allow for the possibility of a competitive atmosphere appearing in such systems. Competitive agents might deliberately give misleading or false information. Even if an agent wants to be honest and give a true response, the VC mechanism

may encourage the selected agent to give a deceptive or dishonest response.

To explain this point, consider the following example. Suppose that a recognizer agent called `Fan` asks an agent called `Thermo` the following question: "Am I allowed to open the window?" (or "Is the opening-window sequence of events prohibited?"). From these kinds of questions, agent `Thermo` could deduce that agent `Fan` wants to open the window. If this act (opening-window) complies with the objectives of agent `Thermo`, then it might respond that this act is not prohibited, even though it is. As a result, `Thermo` can accomplish its objective while deflecting the punishment to the `Fan` agent. In other words, `Thermo`'s response will conform with its objectives instead of conforming with the society's norms.

As a result, the VC is not suitable for open multi-agent societies. To overcome the limitations of the VC discussed above, we propose modified verification component (MVC) so that instead of the recognizer agent asking agents whether a candidate norm (sequence of actions) is a norm or not, the recognizer agent asks other agents whether they can perform a sequence of actions on its behalf. By using this type of question, agents avoid explicit communication about the norm concept.

The idea of a more general VC comes from the fact that all agents, regardless of their architectures, designers, owners or types, must exhibit some behaviours in order to participate in a society, even though their internal architecture may be unknown. In this work, we assume that agents communicate using the Foundation for Intelligent Physical Agents (FIPA)'s Agent Communication Language (ACL)(FIPA, 2002a). We also assume that agents have the capability to perform acts that are relevant to a particular society. However, agents do not have to provide information about their internal states, such as what norms they believe.

Our MVC implementation is modeled using the standard FIPA's Contract Net Interaction Protocol (FIPA, 2002b). The recognizer agent represents the initiator who sends the call for proposal (CFP) communicative acts. In our approach, the recognizer agent does not need to accept any proposals, since it does not need the action to be performed. It needs just to know if other agents will accept to do the action which is prescribed in the CFP. As Figure 3.3 illustrates, the MVC is composed of two steps.

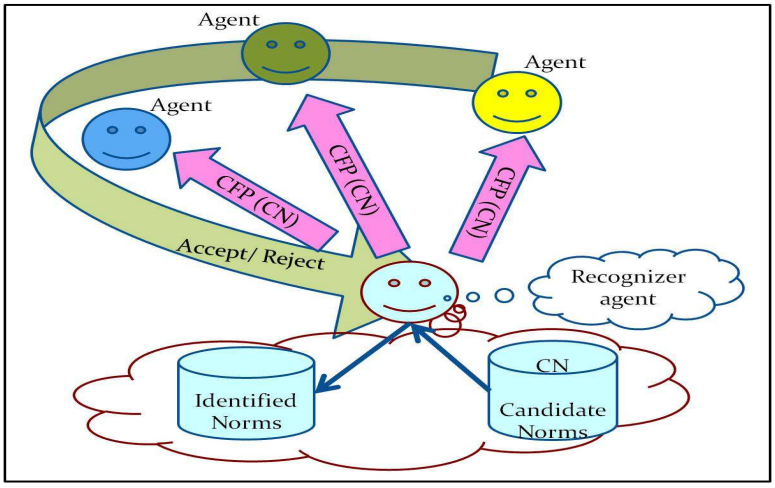


Figure 3.3: The modified verification component (MVC). The recognizer agent picks a candidate norm (CN) from the candidate norms. It then sends a call for proposal (CFP), to perform CN, to a particular number of agents. If the majority response is affirmative, then CN is stored as an identified norm.

**Step 1:** *Selecting agents to be asked.* Particular agents are selected in order to send them a CFP (Algorithm 8, Line 2). We define `NumofWiseAgents` to keep the number of agents that the recognizer agent is going to ask. The CFP requests a proposal for performing a sequence of events (which is effectively the candidate norm). Selecting more than one agent with respect to the same sequence of events mimics the approach of the original VC (Savarimuthu et al., 2010a) by attempting to mitigate

misleading information. There are different possible strategies for selecting which agents to ask. Agents can be selected based on some measure of trust or reputation, they can be preferentially selected because of their long-term participation in a society (and hence being more acclimated to norms), or, as in the case of our experiments, the agents can be selected randomly.

**Step 2:** *Contacting the selected agents.* The recognizer agent sends a CFP to the selected agents (Algorithm 8, Line 4). The decision to accept the selected candidate norm is based on the majority response to the CFP. In the case of candidate prohibition norm verification, if the recognizer agent receives negative proposals (e.g., "refuse") from the majority of the selected agents, then the candidate prohibition norm will be identified as a prohibition norm; otherwise, no norm is identified (Algorithm 8, Lines 5-9). In the case of candidate permission norm verification, if the recognizer agent receives affirmative proposals from the majority of the selected agents, then the candidate permission norm will be identified as a permission norm; otherwise, no norm is identified (Algorithm 9, Lines 5-9).

To illustrate the MVC, consider a book-trading-system that is composed of seller, buyer and shipper agents. Suppose that the recognizer agent wants to verify that buying the book "Big-River-Big-Sea-Untold-stories-of-1949" is permitted. The recognizer agent will send a CFP to the buyer agents asking them to submit their proposals for buying the book. Using FIPA's SL content language (FIPA, 2002c), one of FIPA's possible communicative acts, the CFP reads as follows:

```
(cfp
:sender (agent-identifier :name RecoginzerAgent)
:receiver (set (agent-identifier :name Buyer))
```

:content

“((action (set (agent-identifier :name Buyer)

(Buy Book Big-River-Big-Sea-Untold-stories-of-1949))

))”

:ontology Book-Trading

:language fipa-sl)

The incentive for the asked agents to reply to the recognizer agent's CFP comes from the semantic of the Contract Net interaction protocol. When the asked agents receive CFP, and they can perform the embedded sequence of actions in the CFP, they reply to the verifier agent with their acceptance to do the task and their benefits they want in return. In all cases the verifier agent replies by reject proposal, because it was just inquiring and it was never intending for other agents to follow-through with the action. According to the Contract Net interaction protocol, asked agents will reply with refuse or propose. However, for some reason if the asked agents ignore the sent CFP then the verifier agent ignores the candidate norm (see Algorithm 8 line 8 and Algorithm 9 line 8).

## 3.5 The modified dynamic verification component (MDVC)

Since norms are subject to change, agents need to be able to adopt new norms or repeal existing ones. In order to allow the VC to dynamically infer and recognize the evolution of society's norms, we proposed an alternate version of the VC called

**Algorithm 8** Modified prohibition norm verification

```
1: function MoProhNrmVerifi(SCPhN)
2:     Select NumofWiseAgents agents
3:     for all α ∈ SCPhN do
4:         Send CFP to the selected agents. /*the content of the CFP is the sequence
       of actions α*/
5:         if the majority proposals were negative then
6:             ProhNrmBase← ProhNrmBase ∪ α
7:         else
8:             do nothing
9:         end if
10:    end for
11: end function
```

**Algorithm 9** Modified permission norm verification

```
1: function MoPerNrmVerifi(SCPmN)
2:     Select NumofWiseAgents agents
3:     for all α ∈ SCPmN do
4:         Send CFP to the selected agents. /*the content of the CFP is the sequence
       of events (α)*/
5:         if the majority proposals were affirmative then
6:             PermNrmBase← PermNrmBase ∪ α
7:         else
8:             do nothing
9:         end if
10:    end for
11: end function
```

**Algorithm 10** Modified dynamic prohibition norms verification

```
1: function MoDynProhNrmVerifi(SCPhN)
2:     Select NumofWiseAgents agents
3:     for all α ∈ SCPhN do
4:         Send CFP to NumofWiseAgents agents.
   /*the content of the CFP is the sequence of actions (α)*/
5:         if the majority proposals are negative then
6:             for all Pn ∈ PermNrmBase do /*where Pn is a permission norm*/
7:                 if Pn = α then
8:                     PermNrmBase ← PermNrmBase \ Pn
9:                 end if
10:            end for
11:            ProhNrmBase← ProhNrmBase ∪ α
12:        else
13:            do nothing
14:        end if
15:    end for
16: end function
```

**Algorithm 11** Modified dynamic permission norms verification

```
1: function MoDynPerNrmVerifi(SCPmN)
2:     Select NumofWiseAgents agents
3:     for all  α ∈ SCPmN do
4:         Send CFP to NumofWiseAgents agents.
   /*the content of the CFP is the sequence of actions (α)*/
5:         if the majority proposals were affirmative then
6:             for all Fn ∈ ProhNrmBase do /*where Fn is a prohibition norm*/
7:                 if Fn = α then
8:                     ProhNrmBase ← ProhNrmBase \ Fn
9:                 end if
10:            end for
11:            PermNrmBase← PermNrmBase ∪ α
12:        else
13:            do nothing
14:        end if
15:    end for
16: end function
```

the dynamic verification component (DVC) (see Section 3.3). This alternate version enables the consistency between prohibition and permission norms to be preserved.

In this section, we apply a further modification to the DVC, much like the modification presented to the VC in Section 3.4. Let us call the new resulting component the modified dynamic verification component (MDVC). Similar to the MVC, in the MDVC, the recognizer agent sends to a particular number of agents a CFP to perform a sequence of actions which forms a candidate norm.

In the case of verifying a candidate prohibition norm, if the majority of the received proposals are negative, then the candidate prohibition norm is matched against the permission norms (Algorithm 10, Lines 5-14). If it does not match, the candidate prohibition norm is stored in `ProhNrmBase` (Algorithm 10, Line 11). However, if the candidate prohibition norm matches a permission norm, the matched permission norm is deleted from `PermNrmBase` and the candidate prohibition norm is added to `ProhNrmBase` (Algorithm 10, Lines 7-9).

In the case of verifying a candidate permission norm, if the majority of the received proposals are affirmative, the candidate permission norm is matched against the prohibition norms (see Algorithm 11, Lines 5-14). If it does not match any of the prohibition norms, then it is stored in `PermNrmBase`. If it matches a prohibition norm then the matched prohibition norm is deleted from `ProhNrmBase` and the candidate permission norm is added to `PermNrmBase` (Algorithm 11, Lines 7-11). Therefore, the recognizer agent will not have an action defined as a prohibition and as a permission norm at the same time. As a result of the operations of the DVC, the recognizer agent maintains consistency among the permission and prohibition norms.

## 3.6 Summary

In this chapter we developed an agent architecture and algorithms to identify prohibition and permission norms. We presented the DVC, a modification on the known VC (Savarimuthu et al., 2010b) as a way of identifying the changes that may happen in a society's norms. We address the limitations of the VC proposed when it is used in open multi-agent societies by proposing modifications to overcome these limitations.

Identifying permission norm along with prohibition norm helps the recognizer agent not only to identify new norms but also to identify the norms' changes. Identifying a society's norms in general helps agents to predict and regulate agents' behaviours, in addition to their role in goal formation and plan adoption. Permission norm identification, in particular, helps newcomer agents to detect the repealed prohibited norms and to interact with the society even if they have not yet inferred the prohibition norms. As a result, the agent does not waste time waiting for sanction events to occur, and it does not lose its resources or utilities by acting recklessly without knowing the norms and risking sanctions. Hence, the agent can match the event it wants to execute with the permission norm base and then execute it without worrying about sanctions or violating the society's norms.

For convenience, in the next chapter we present the experiment results of our work and the comparison results of our work with one of the most popular approaches in norm identification.

# Chapter 4

# Experiment results for norm identification

## 4.1 Experimental setup

In this chapter we describe a restaurant scenario and discuss the results obtained by implementing our proposed architecture and algorithms. We have three main sets of experiments: one to show that our proposed agent architecture and algorithms are able to identify repealed and new permission and prohibition norms. In the second set of experiments, we demonstrate the significance of permission norms in detecting the repealed prohibition norm by comparing our approach with Savarimuthu et al. (2010a) approach. In the other set of experiments, we compare the VC as described in (Savarimuthu et al., 2010a) to our proposed MVC. Then, we test our proposed MDVC in a scenario involving norms change.

Java JDK 1.6.0 within JADE-3.7 (Bellifemine et al., 2007) software agent development environment is used to build a restaurant interaction scenario as an example. Our experiments are based on a society of 22 agents. The parameters are set in the experiments as follows:

- `QueueSize`: 8 events.
- `RegThresh`: 8 regular events.
- `SanThresh`: 1 sanction events.
- `NumofWiseAgents`: 3 agents are selected randomly.

We selected a queue size of 8 events based on several experiments with queues of various sizes (256, 128, 64, 32, 16 and 8). Increasing the size of the queue for this scenario does not give more information for the recognizer agent to identify the prohibition norms, unless the norms are composed of more than 8 events.

Our agent observes other agents' events and store them in their corresponding queues. The first added event leaves the queue if the queue is full when the most recently observed event needs to be added. In our experiments, the recognizer agent maintains a queue for each agent in the joined society. The number of events that the recognizer agent is able to store depends on the size of the queue which itself depends on the characteristics of the system being modeled. If the society has norms that are composed of `n` actions, then the size of the queue should be at least `n+1` in order to show the sanction event.

The `RegThresh` parameter was chosen to equal the queue size. This way, when the queue is full of regular events, the permission norm identification algorithm is called.

The prohibition norm identification algorithm is called every time the recognizer agent observes a sanction event. Because of this, we set the `SanThresh` parameter to one.

Our restaurant society consists of customer agents and a restaurant supervisor agent. The restaurant has the following events: *Reserving, Eating, Dropping, Paying, Smoking, Tipping, Yelling*, and *Leaving*. The supervisor agent is the police agent in the restaurant and plays the norm enforcement role. Any customer who violates the norms of the restaurant is sanctioned by the supervisor agent. We assume that the supervisor agent is already aware of the prohibition norms of the society, which could be one event or a sequence of events.

In our experiments, *Dropping*, *Smoking* or *Yelling* are defined as prohibition norms. Customer agents are able to execute any event in the restaurant as many times as they want, and they are sanctioned if they violate the restaurant society's norms.

## 4.2 Experiment set 1 - Newcomer agent scenario

The aim of this experiment is to show that our proposed architecture and algorithms are able to infer changeable prohibition and permission norms. In this scenario, when a newcomer agent joins the restaurant society and before he acts in the society, he will try to infer the norms that govern the joined society. We assume that the society has some disobedient customer agents who sometimes violate the norms. The restaurant society in this experiment consists of one supervisor agent, one recognizer agent and twenty customer agents which continuously performed random events. Note that we ignore the semantic problems in the events that a customer agent may perform. For

example, in our scenarios the customer may Eat, Reserve, Leave, Eat and then Pay, in an order that is not semantically rational.

The results of the experiment, shown in Figures 4.1 and 4.2, demonstrate that the newcomer agent (the recognizer agent) is able to recognize and identify the society's prohibition norms *(Dropping (D), Smoking (S), Yelling (Y))* and some of the permission norms. We further see that the success of norm identification increases as the events executed in the society increase.

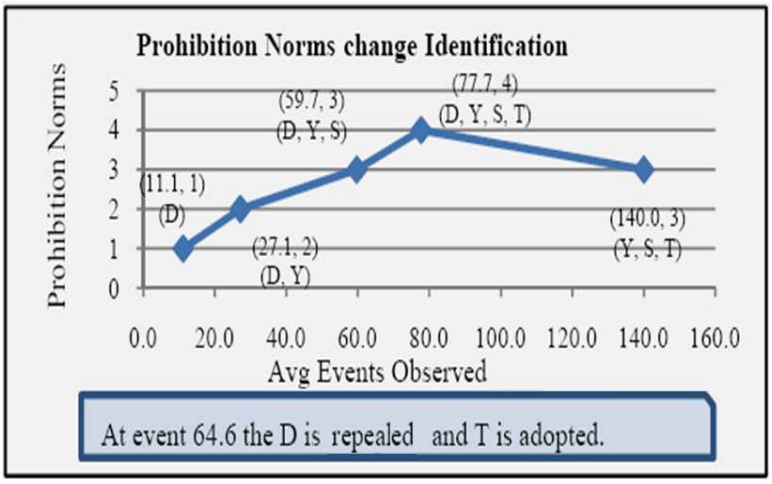


Figure 4.1: Prohibition norms change identification

After the recognizer agent has identified the prohibition norms, the norms of the restaurant society are changed. A new prohibition norm, *Tipping*, is adopted and a known prohibition norm, *Dropping*, is repealed. The supervisor agent and the other customer agents are aware of this change. Our recognizer agent needs to detect this change.

The results in Figures 4.1 and 4.2 demonstrate how the recognizer agent, who applies our proposed architecture and uses the DVC algorithm, can identify the changed

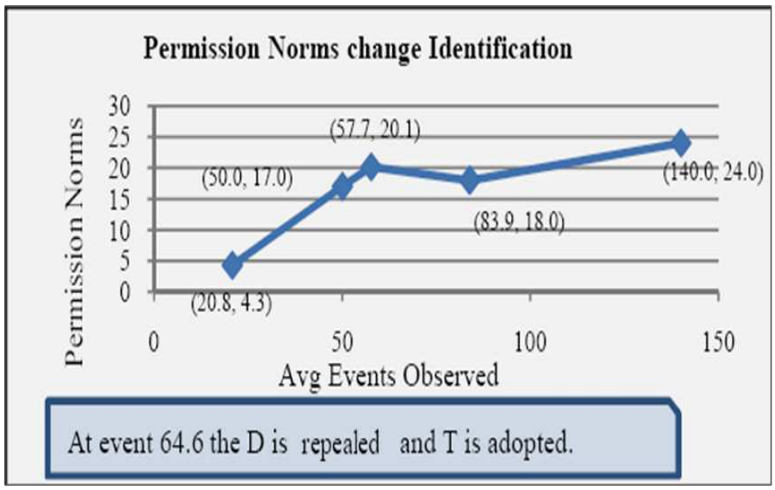


Figure 4.2: Permission norms change identification

norms. As we see in Figure 4.1, at event 77.7, the new adopted norm, *Tipping*, is identified before identifying that the prohibition norm, *Dropping*, is repealed. Thus, at event 77.7, the recognizer agent has four prohibition norms while the society has only three. At event 140, the recognizer agent identifies that the prohibition norm, *Dropping*, has been repealed.

In Figure 4.2, as a result of adopting the norm *Tipping*, the number of permission norms is decreased at event 83.9. The number of permission norms is affected by the identification of prohibition norms. Hence, at event 83.9, the recognizer agent loses some permission norms. Figure 4.2 shows that the number of identified permission norms increases as the number of executed events increase. At event 140, the recognizer agent identifies that the prohibition norm, *Dropping*, has been repealed. Consequently it identifies more permission norms.

## 4.3 Experiment set 2 - Newcomer agent scenario

The aim of this experiment is to demonstrate the significance of using permission norms in detecting the repealed prohibition norms. We compare our approach which utilizes the permission norm for detecting the repealed prohibition norm with Savarimuthu et al. (2010a) approach. According to Savarimuthu, a prohibition norm is repealed if that prohibition norm has not been detected as a prohibition norm for a particular time. In contrast, our approach detects that a prohibition norm is repealed only if it is identified as a permission norm. The problem with the former approach is that if a particular prohibition norm has not been violated for a particular time then this norm will not be inferred by the norm identification algorithm; as a result it will be repealed from the agent belief base even if the norm still exists in the society.

In this scenario we compare two agents: one that utilizes the permission norm and another that uses only the prohibition norm. The supervisor agent and the other customer agents are aware of the society norms, which are *(Dropping, Smoking, Yelling)*.

In Savarimuthu et al. (2010a)'s approach the period of time during which the agent needs to wait before repealing a prohibition norm in case no violation recorded for that norm, plays a crucial role in norm identification. Let us call this period `calmTime`. In our experiments, setting `calmTime` to 10 means that if the prohibition norm identification algorithm is triggered 10 times and a prohibition norm has not been inferred then that prohibition norm is repealed from the agent belief base.

Figures 4.3 through 4.5 show the comparison results between the two approaches; our approach is the one with permission norm and the other approach is the one

without permission norm. At event 45, a new prohibition norm, *Tipping*, is adopted and a known prohibition norm, *Dropping*, is repealed. The supervisor agent and the other customer agents are aware of this change. We choose event 45 to change the norms since we find based on experiments 44.6 events in average are enough for both agents to infer the three prohibited norms. Figure 4.3 shows the result when `calmTime` equal to 10. We notice from the Figure that the agent that does not use permission norm falls in the trap of mistakenly repealing prohibition norms. Figures 4.4 and 4.5, which use `calmTime` values of 25 and 50, respectively, show that as we increase the `calmTime` the falling in that trap is decreased. However, as we increase the `calmTime` the agent needs to observe more events to detect a repealed norm (see Figure 4.5). In contrast, the agent that utilizes the permission norm to detect the repealed norm is required to observe fewer events; also he does not suffer from the trap of mistakenly revoking a prohibition norm.

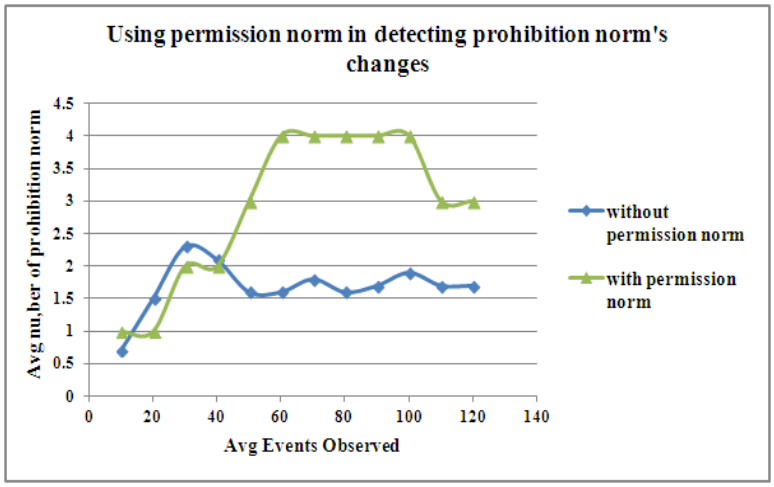


Figure 4.3: Detecting norm changes using permission norm with `calmTime` = 10.

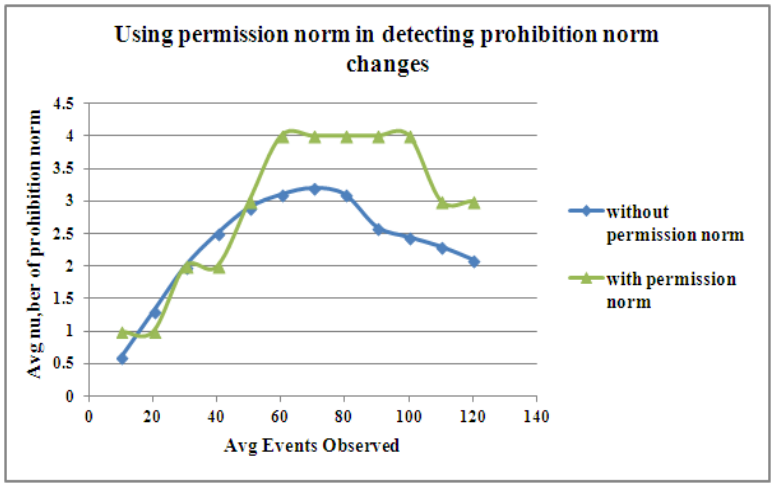


Figure 4.4: Detecting norm changes using permission norm with `calmTime` = 25.

## 4.4 Experiment set 3 - Newcomer agent scenario

As a complement to comparing our norm identification approach with Savarimuthu et al. (2010a), in this experiment we show that the VC is not adequate for heterogeneous multi-agent systems. Therefore, we compare it with our proposed MVC and MDVC.

In this experiment setup, the customer agents form a heterogeneous society. Based on the heterogeneity in the society, it is possible that some agents are not aware of the concept of norm. The restaurant society in this experiment consists of two groups of customer agents, one supervisor agent and our recognizer agent. The first customer group consists of ten agents who are aware of the concept of norm. The second group of ten agents has a different internal architecture and the agents are not aware of the concept of norm. In this scenario we assume that agents use the FIPA's Agent Communication Language (ACL).

In these experiments, we compare the VC and the MVC relative to the efficiency

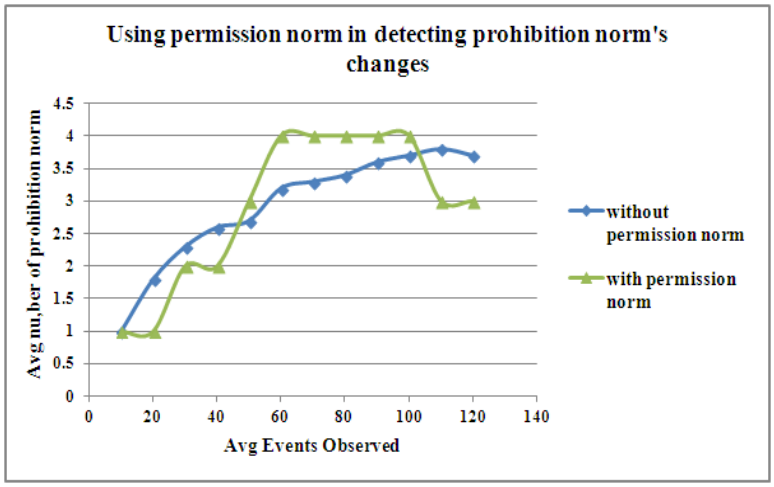


Figure 4.5: Detecting norm changes using permission norm with `calmTime` = 50.

of inferring prohibition norms. The component that infers norms using the smaller number of observed events is considered to be more efficient.

In this scenario, 50% of the customers do not have a concept of norm. This affects the VC and the MVC differently. When the VC is applied, selected agents are asked about norms directly. This means that only 50% of the agents are capable of replying (since the other 50% do not have a concept of norm). Even worse, if we assume that some agents might not cooperate with the recognizer agent's inquiry then the probability of receiving answers from the customer agents might be less than 50%. In contrast, when the MVC is applied, selected agents are asked about behaviours instead of norms. Hence 100% of the agents are able to reply, regardless of whether they are aware of the concept of norm or not. Using CFP for communication encourages other agents to reply since they expect a benefit by performing the task.

The experiment was repeated ten times. In each execution, the number of observed events required to identify norms was recorded. The average number of observed events was then calculated for the ten executions. The experiment results show

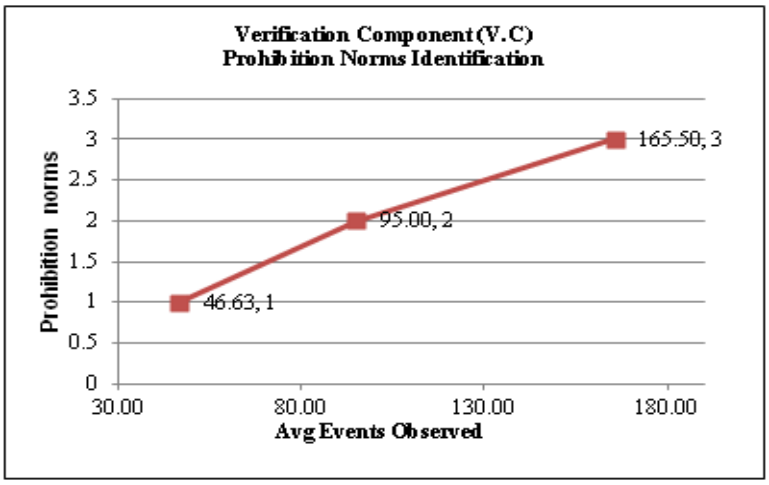


Figure 4.6: Verification component (VC) for prohibition norm identification. After the observer agent witnessed 46.63 events (in average), it inferred one prohibition norm. It inferred another at event 95.00, and at 165.50 it had inferred the last prohibition norm in the society.

that the newcomer agent infers all restaurant prohibition norms *(Dropping, Smoking, Yelling)* by event 165.5 (on average) when the VC is applied (see Figure 4.6) and by event 43.4 (on average) when the MVC is applied (see Figure 4.7). Figure 4.8 shows that the MVC needs fewer events to infer the prohibition norms than the VC.

Testing the MDVC begins after the newcomer agent has identified all prohibition norms in the restaurant society. After event 44, the restaurant supervisor changes the society's norms, by adopting a new prohibition norm, *Tipping*, and repealing another prohibition norm, *Smoking*.

The experiments described in Figures 4.9 and 4.10 express the results of identifying the changed norms for the customer agent who uses the MDVC. As we see in Figure 4.9, at event 62, the newly adopted norm *Tipping* is identified before the repealing of *Smoking* is identified. At event 92, the recognizer agent identifies the repealed prohibition norm *Smoking* and loses the prohibition norm *Dropping*. At event

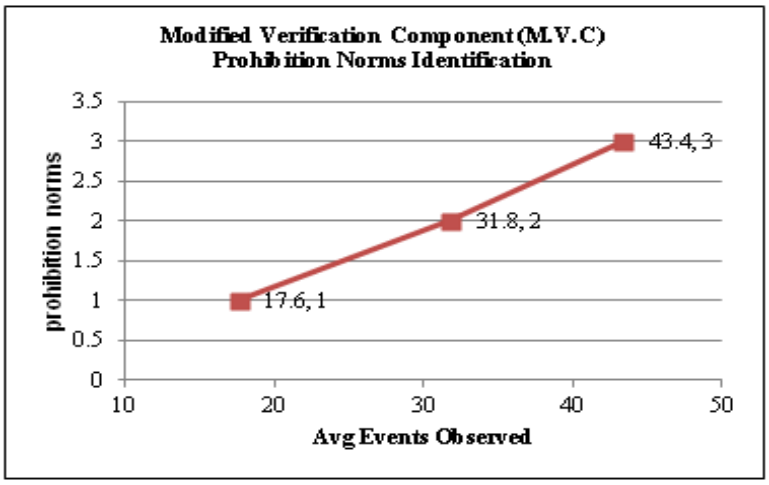


Figure 4.7: Modified verification component (MVC) for prohibition norm identification. The observer agent inferred one prohibition norm after 17.60 events were observed. It inferred another at event 31.80 and at 43.4 it had inferred the last prohibition norm in the society.

110, *Dropping* is again identified as a prohibition norm. Indeed, this point shows how the MVDC preserves consistency among norms. In Figures 4.9 and 4.10 at events 92 and 110, we see that as the number of identified prohibition norms decreases, the number of identified permission norms increases, and vice versa.

## 4.5 Experiment discussion

In this chapter we present the experiment results of our work in dynamic norm identification and we compare our work with Savarimuthu et al. (2010a), one of the most popular work in norm identification. To demonstrate our work, we create a restaurant society scenario using JADE software agent development framework, and apply our proposed architecture and algorithms. We show how our proposed work can infer the changes that might occur to the norms of a society online. In the second set of experiments, we show the significance role of permission norms in detecting the

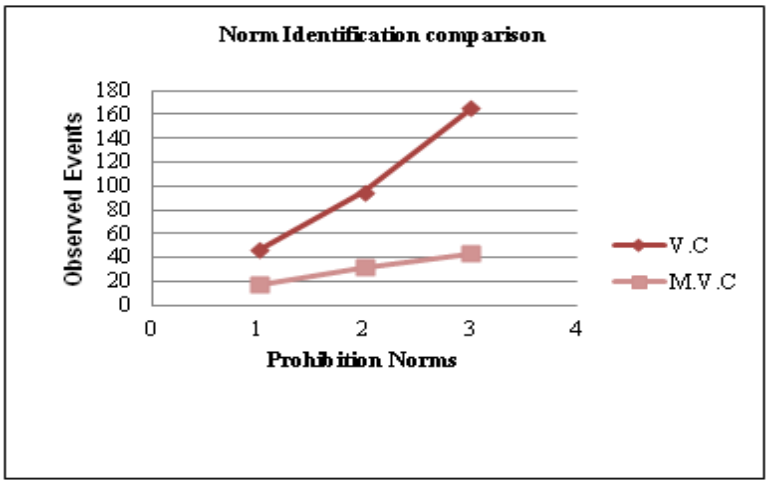


Figure 4.8: Norm identification comparison. The verification component (VC) needs more events than the modified verification component (MVC) in order to infer the society norms.

repealed prohibition norms. In the last set of the experiments, we demonstrate that the MVC can identify norms in open multi-agent societies, even if all agents in the society are not aware of the concept of norm. This overcomes one limitation of the VC, in which the probability of failure in identifying norms increases as the number of agents who are not aware of the concept of norm increases. We also show how the MDVC is able to identify and dynamically infer changes that can occur to the norms of an open multi-agent society. Our agent with this mechanism, however, fails to choose the best behaviour to achieve its objectives and avoid norms violation. In the next chapter, we overcome this limitation by developing a mechanism for that purpose.

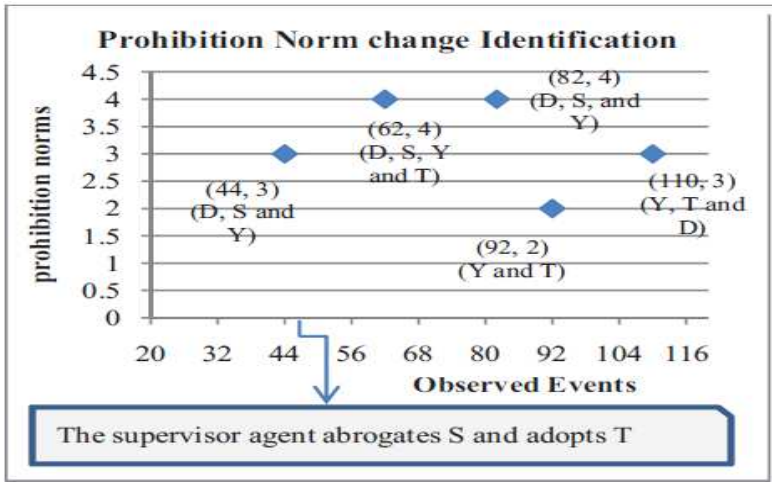


Figure 4.9: Prohibition norm change identification using MDVC, D, S, Y and T represent Dropping, Smoking, Yelling and Tipping, respectively. When 44 events were perceived, there was no change to the prohibition norm base. After event 44 the restaurant norms are changed by revoking S and adopting T. At event 62 the agent infers the extra norm T. At event 92, the agent detects the revoked prohibition norm S. This means that the agent adopts the new norm T before revoking the old one S.

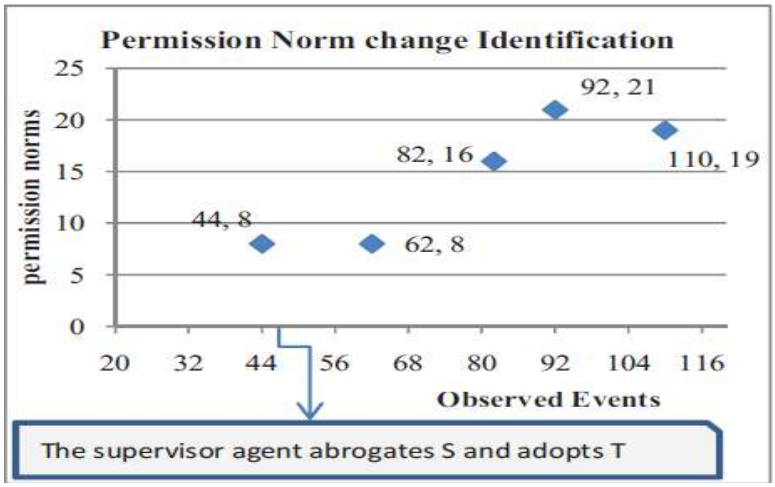


Figure 4.10: Permission norm change identification using MDVC. The number of permission norms is decreased by increasing the number of prohibition norms, as seen at event 92 and 110. Also the figure shows that the number of identified permission norms is increased by increasing the number of executed events.

# Chapter 5

# Norm Representation and Reasoning: A Formalization in Event Calculus

In open multi-agent systems, heterogeneous agents are designed by various people. Agents enter and leave the system while acting autonomously towards self-interested goals (Ramchurn et al., 2004). In such environments, norms play an important role for regulating and predicting an agent's behaviour.

Electronic Institutions (Esteva et al., 2001) is an approach which assumes that agents always comply with and follow the norms. Although this makes their behaviour more predictable and coordinated, this drastically decreases agents' flexibility and autonomy. Among self-interested agents, their highest priority is achieving their own goals. Therefore they should not mindlessly obey norms unless they benefit from such obedience. Thus, most researchers in normative multi-agent systems adopt an

alternative view in which norms are considered as soft constraints and agents have the choice to comply with or to violate norms (Aldewereld et al., 2007).

Similar to human societies, agents that violate norms are subject to sanctions. Sanctions have been used as a norm enforcement tool to urge agents to respect norms and keep the system stable (Castelfranchi, 2003). Thus we need agents to be able to adapt their behaviour according to their society's norms to avoid potential punishments. We face at least two challenges in building such agents. The first challenge is determining how agents know the prevalent norms. The second is creating an application-independent mechanism to help agents consider dynamic norms in their practical reasoning. This is required because norms are subject to change or emerge at runtime and because agents can travel to different societies that have different norms.

In order to address these two challenges, we need to design reasoning mechanisms able to identify dynamic norms as well as consider these norms during an agent's practical reasoning. In Chapter 3, we address the former challenge. In this chapter, we are concerned with the latter challenge, and focus on reasoning mechanisms for flexible norm compliance.

Here we are not concerned with modeling normative environments or describing the behaviour of multi-agent systems. Rather, we are concerned with the impact of norms on an individual agent's behaviours at runtime. We propose a norm's formal representation using event calculus (EC) and an application-independent mechanism for reasoning about the effects of norms on an agent's behaviour. The mechanism we develop in this Chapter allows an agent to choose the set of best plans among multiple applicable plans, and choose the safest plan among the best plans. Where

we define the best plan as the plan with the highest positive utility for that agent, and safest plan as the plan that complies with the maximum number of permission norms.

The remainder of this chapter is organized as follows. We first present the limitations of classical EC and propose a modification to mitigate these limitations (Section 5.1). We then propose a formal representation of norm (Section 5.2). In 5.3 we propose a normative reasoning mechanism. We discuss the significance of using the permission norm in normative reasoning mechanism in Section 5.4. Our normative reasoning mechanism using the permission norm presented in (Section 5.5). We conclude the chapter with a summary (Section 5.6).

## 5.1 Representing delayed or immediate effects in event calculus

Suppose that in a particular trading system, if an agent orders an item from system x, it will receive the item after two days, but if the order is from system y then it will receive it after three days. To represent this using EC, `order(Item,System)` is an action to order an item from a system and `received(Item)` is a fluent indicating whether the item is received or not. To represent that using EC we have:
`initiates(order(item,x),received(item),T1)` and
`initiates(order(item,y),received(item),T1)`.
Both representations mean that the item will be received after the order is set. However, EC fails to represent the delay of the effects of the actions. In other words, the representation above does not reveal the difference between the effects of the actions.

One of the basic predicates of EC is `initiates(A,F,T)`, which states that the occurrence of action `A` at time `T` leads fluent `F` to be true at the next time step after `T`. This effectively means that the norm starting time point is after `T`. However, this is not always the case. Indeed, there are at least two other possibilities:

1. The action's effects may take place at the time of the action's occurrence. Hence, a fluent becomes true at the same time as the action occurred (e.g., once you enter a library it is prohibited to speak loudly) (Hashmi et al., 2014).

2. The action's effects may take place a while after the action occurred. Hence, a fluent becomes true at time greater than the time of the action's occurrence (e.g., while it is prohibited to use cell phones in an airplane, this norm is not in force as soon as the passenger enters the airplane, but rather after the passenger puts his handbag in the cabinet and sits down).

In order to be able to represent the additional cases, we replace the EC predicate `initiates(A,F,T)` with the predicate `initiatesAt(A,F,T1,T2)`, which states that the occurrence of action `A` at time `T1` will make fluent `F` true at `T2`.

The same argument applies for the EC predicate `terminates(A,F,T)`, but here action `A` triggers fluent `F` to be false. We replace this predicate with `terminatesAt(A,F,T1,T2)`, which states that the occurrence of action `A` at time `T1` will make fluent `F` false at time `T2`. Consequently, we need to modify basic EC axioms (see Section 2.4). The extended axioms important to our work are as follows:

`EC1′:clipped(T1,F,T3)`$\leftarrow$
`happens(A,T2)` & `terminatesAt(A,F,T2,T3)` & `T1`$<$`T2` & `T2`$\leq$`T3`

This states that fluent F is terminated by the occurrence of action A between times T1 and T3.

EC3′:holdsAt(F,T2)← happens(A,T1) & initiatesAt(A,F,T1,T2) &
T1≤T2 & ¬clipped(T1,F,T2)

This states that fluent F holds at time T2 if action A occurred at time T1, fluent F became true at time T2, and F has not been terminated between T1 and T2.

We further define the predicate between(A,T1,T2). This states that action A occurred after time T1 and before T2. We use this predicate to formally represent norms.

## 5.2 Norm representation

In order to employ norms in an agent's practical reasoning, a formal representation is needed. Our view of norms follows Anderson's reduction view (Anderson, 1958) which states that norm violation is necessarily followed by a sanction (Soeteman, 2001). We use sanctions and rewards events as flags to refer to norm-violation and norm-compliance. Thus, in our norm representation we define fluents to represent those flags. Using EC, we represent norms as a sequence of actions that make fluents true. We make the pivot of our norm representation to be the fluents (sanctions and rewards). This is because our agent's decision to comply or violate norms is based on sanctions and rewards. The punishment or reward represents an incentive for agents to change their behaviours. For this purpose we introduce the following fluents:

`fPun(Nid,S)` is a fluent which becomes true if the prohibition norm `Nid` is violated. The sanction of the violation is `S`. `Nid` is a unique number of prohibition norm, where `S` is an integer representing the sanction value.

`oPun(Nid,S)` is a fluent which becomes true if the obligation norm `Nid` has not been fulfilled. The punishment issued for this violation is `S`. `Nid` is a norm identification number and `S` is the punishment value.

`oRew(Nid,R)` is a fluent which becomes true if the obligation norm `Nid` has been fulfilled. The variable `R` refers to the reward value.

Sanctions and rewards play a fundamental role in representing norms. A sanction refers to a norm violation event, and a reward refers to an obligation norm fulfillment. Using the above three fluents `fPun(Nid,S)`, `oPun(Nid,S)` and `oRew(Nid,R)`, we can now formally define prohibition and obligation norms.

### 5.2.1 Definition of prohibition norm

Following our view of norms Section 2.1.1, the formal definition for prohibition norms is as follows:

`initiatesAt(An,fPun(Nid,S),Tn,Tn+1):- C, happens(A1,T1)` & $\cdots$ &
`happens(An,Tn)` & `T1<T2<` & $\cdots$ & `<Tn`.

This representation of prohibition norm contains the following parts. `D`, the deontic type, is a prohibition norm; on the left hand side of the definition we use the `fPun` fluent which refers to a prohibition norm violation state. `C` is the norm's

context. Seq is a sequence of actions, A1,A2,$\cdots$,An, that is prohibited. S is the sanction value which will be applied on the violator agent. R is empty for prohibition norm.

Our prohibition norm representation states that: if actions A1, $\cdots$ ,An occurred at time T1, $\cdots$ ,Tn respectively, and the context C was a logical consequence from the agent's belief base, then the sanction that might be applied on the actor after Tn is S. If the order of actions is not important in a norm, then we omit the dependencies among T1,T2, $\cdots$ ,Tn-1. However T1,T2, $\cdots$ ,Tn-1 should be less than Tn. For example, if performing actions X, Y and Z in any order is prohibited then we do not need to specify which action occurs before or after which action.

### 5.2.2 Definition of obligation norm

Following our view of norms Section 2.1.1, we can now give below the formal definition for obligation norm.

Let $\alpha$ be a sequence of actions, possibly empty, and Seq be a sequence of prescribed actions that is supposed to be performed by an agent. An obligation norm violation occurs if in a particular context (which $\alpha$ is a part of), Seq does not occur. Fluent oPun(Nid,S) becomes true if the obligation norm Nid is violated. The obligation norm fulfillment occurs when the Seq occurs in that context. Fluent oRew(Nid,S) becomes true if the obligation norm Nid is fulfilled.

We represent obligation norms by two rules. The first of these rules is as follows:

initiatesAt(Ai,oPun(Nid,S),Ti,Tn+1):- $\beta$ & happens(A1,T1) & $\cdots$ &

happens(Ai,Ti) & $\cdots$ & $\neg$happens(Aj,Tj) | $\cdots$ | $\neg$happens(An,Tn)
& T1< & $\cdots$ & <Ti< & $\cdots$ & <Tj< & $\cdots$ & <Tn.

This representation contains the following parts. D, the deontic type, is an obligation norm; this is so defined because we use the oPun fluent in the left-hand side of the definition, which refers to an obligation norm violation state. C is the norm's context, composed of $\beta$ and $\alpha$, where $\beta$ represents the world's states of the context and $\alpha$ = A1,A2,$\cdots$,Ai. Seq is a sequence of actions (Aj,Aj+1,$\cdots$,An) that is obliged to be performed. S is the sanction value which will be applied on the violator agent.

The first rule states that, if a part of the context C ($\beta$) is a logical consequence from the agent belief base, and a (possibly empty) sequence of actions, $\alpha$, A1,A2,$\cdots$,Ai occurs at time T1,T2,$\cdots$,Ti respectively, and a sequence of actions (Seq) Aj,Aj+1,$\cdots$,An does not occur at Tj,Tj+1,$\cdots$,Tn, then the sanction that may be applied after Tn is S.

In addition to the first rule we need the following rule only if the obligation norm fulfillment is subject to a reward.

initiatesAt(An,oRew(Nid,R),Ti,Tn+1):- $\beta$ & happens(A1,T1)& $\cdots$&
happens(Ai,Ti) & $\cdots$& happens(Aj,Tj) & $\cdots$ & happens(An,Tn)
& T1< & $\cdots$ & <Ti< & $\cdots$ & <Tj< & $\cdots$ & <Tn.

This second rule states that, if a part of the context C ($\beta$) is entailed from the agent's belief base, and a (possibly empty) sequence of actions ($\alpha$) A1,A2,$\cdots$,Ai occurs at time T1,T2,$\cdots$,Ti, and a sequence of actions (Seq) Aj,Aj+1,$\cdots$,An occurs at Tj,Tj+1,$\cdots$,Tn, then the reward that might be granted after Tn is R.

We will illustrate the operation of these rules with two examples.

**Example 5.2.1.** In a particular auction, if an agent bids and wins, then within 24 hours it should pay for the item and fill a survey. If the agent does this, it will receive a discount code of $5 on its next purchase; otherwise, the agent will be added to the auction's black list.

In this example a violation occurs if the agent performs the action "bid" but does not perform the actions "pay" or "fill-survey". The context `C` is composed of:

$\alpha$ = `happens(bid,T1)` and $\beta$ = `holdsAt(win,T2)`.

The obliged sequence of actions (`Seq`) is:

`happens(pay,T3)` | `happens(fill-survey,T4)`
such that `T1<T2<T3<T4`.

Assume that, for our agent, the punishment value is equivalent to $10. This obligation norm is then represented as follows:

```
initiatesAt(bid,oPun(1,$10),T1,T4):- happens(bid,T1)
& holdsAt(win,T2) & ¬happens(pay,T3) | ¬happens(fill-survey,T4)
& T1<T2 & T2<T3 & T3<T4.

initiatesAt(bid,oRew(1,$5),T1,T4):- happens(bid,T1)
& holdsAt(win,T2) & happens(pay,T3) & happens(fill-survey,T4)
& T1<T2 & T2<T3 & T3<T4.
```

As we can tell from the example above, norms can be composed of several ac-

tions, which are `pay` and then `fill-survey`. In this example the order of the actions is important; however, if the order of the actions is not important we omit the dependencies among the times of the occurrence actions. In the next example, we consider a situation in which no actions are given in the norm's context, and, because no reward is given, only one rule is required to define the norm.

**Example 5.2.2.** In a particular society, the obligation norm is "At Christmas you should call your parents, visit them and give them a gift. Otherwise they will feel disappointed". Here, Christmas is the norm's context. The person should do three actions to fulfill this obligation norm: "call", "visit" and "give". The punishment of violating this norm is parents' disappointment.

In this example, the actions part of the context ($\alpha$) is empty (in this case the first argument of the predicate `initiatesAt` is not used), where

`Seq` = ¬`happens(call,T1)` | ¬`happens(visit,T2)` | ¬`happens(give,T3)`
such that `T1<T2<T3`.

Assuming that there are no rewards of this norm's fulfillment, then we only need one rule to represent the norm:

`initiatesAt(A,oPun(normID, disappointed),T0,T3):-`
`holdsAt(christmas,T1)` & ¬`happens(call,T1)` | ¬`happens(visit,T2)` |
¬`happens(give,T3)` & `T1<T2` & `T2<T3`.

## 5.3 Our normative reasoning mechanism

In this section we present a mechanism that allows our agent to adapt its behaviours according to the norms of a society. In the context of the BDI architecture (see Section 2.2), after the applicable plans are selected, our mechanism utilizes the plans and finds the best one to add to the agent's intention set. In Figure 5.1, an overview of the basic BDI interpreter is illustrated as white boxes while our proposed additions are presented as grey boxes.

Our normative reasoning mechanism not only checks whether a plan respects the norms or not, but also takes the agent's history into consideration. For example, an agent can have two plans, P1 and P2, each respecting norms. However, it is possible that performing P1 followed by P2 can violate norms if P1 has one or more actions that, when combined with one or more actions from P2, violates norms. Our mechanism handles this by enabling the agent to discover such potential violations. For example, suppose that the sanction of running a red light three times is the suspension of a driver's license. If there is no mechanism to take an agent's past actions into consideration (e.g., how many times it has run a red light before), the agent should never predict that he will incur a sanction by running a red light.

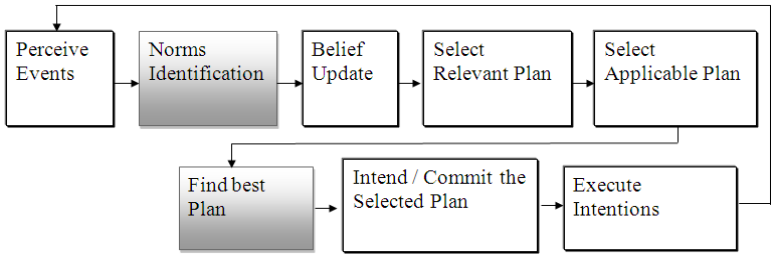


Figure 5.1: Reasoning process flow.

In Section 5.2 we presented how we formalize prohibition and obligation norm using EC. In the following, we prepare for the necessary definitions and axioms for our normative reasoning mechanism.

We introduce the following fluent that states whether a plan has more profits than losses. The fluent `help(Plan)` will be true if the rewards outweigh the punishments of performing `Plan`. The punishments value comes from violating prohibition norms or from failing to fulfill obligation norms (stored in the variables `Losses1` and `Losses2` respectively). The rewards value comes from achieving the goal associated with `Plan` (stored in variable `Points` in `helpful-rule` described below) and fulfilling obligation norms (stored in the variable `Wins` in `helpful-rule`). We define the predicate `goalpreference(G,Points)` to describe the preference of achieving goal `G`, where `Points` is an integer number that refers to the importance of `G`. The goal importance is determined according to the agent's designed objectives.

The default value of our defined fluents is false; their values might be changed after our mechanism does a reasoning about a particular plan. Hence, we need to reset those fluent to the default values after each checking for each plan. In order to terminate fluents, a domain-independent special event * is used as a wild card variable to denote any action. It refers to the fact that the associated fluents become false. For that purpose we define the following domain-independent axioms using the EC framework. Our agents implement these axioms in their normative reasoning mechanism:

- `EC1′` & `EC3′` (see section 5.1)
- `Ax1: between(A,T1,T2) :- happens(A,T)` & `T1 <T` & `T <T2`

- Ax2: `terminatesAt(*,help(P),T1,T2):- happens(*,T1)` & `T1<T2`
- Ax3: `terminatesAt(*,fPun(I,S),T1,T2):- happens(*,T1)` & `T1<T2`
- Ax4: `terminatesAt(*,oPun(I,S),T1,T2):- happens(*,T1)` & `T1<T2`
- Ax5: `terminatesAt(*,oRew(I,S),T1,T2):- happens(*,T1)` & `T1<T2`

These axioms allow agents to find a potential norm violation/fulfillment that can result from executing the current plan. If we also want the agent to be able to find the potential norm's violation/fulfillment resulting from the combination of actions of the current and the previous plan, we need to add another `happens(*,T2)` predicate to the right-hand side of `Ax2` through `Ax5`. Here, the * action monitors the end point of those fluents that are mentioned in the axioms above. For example, `Ax2` now becomes

`terminatesAt(*,help(P),T1,T3):- happens(*,T1)` & `happens(*,T2)` & `T1<T2` & `T2<T3`.

The same addition should be added to `Ax3` through `Ax5`.

In addition to the above domain independent axioms, we introduce the following rule, which we have called `helpful-rule`.

**helpful-rule:**

```
initiatesAt(_,help(Plan),T1,T2):-
.findall(V1,holdsAt(oRew(,V1),T2+1),Wins) &
.findall(V2,holdsAt(fPun(_,V2), T2), Losses1) &
.findall(V3,holdsAt(oPun(_,V3), T2), Losses2) &
goalpreference(G,Points) &
```

```
(Points + sum(Wins)- sum(Losses1) - sum(Losses2)) > 0
```

Using `helpful-rule`, the agent determines whether a plan is helpful. The righ-hand side of this rule becomes true when the left-hand side finds that all the rewards that can result from executing the current plan (the plan under test) are more than the punishment that can result from executing the same plan. The agent asserts the actions of a plan into a temporary belief base, which is a copy of the agent's belief base, using the `happens` predicate in order to simulate that these actions have occurred, (see Algorithm 12, line 4, and Algorithm 13).

Generally, predicate `.findall(V,holdsAt(p(_,V),_),Set)` (as in Prolog and Jason (see Section 2.2.1)) obtains all the values of `V` where fluent `P` is true. `V` represents the value of a punishment if `p` is `fPun` or `oPun`, or a reward if `p` is `oRew`. Variable `Set` unifies with the set of `V` values. Finally, `Wins` obtains the rewards that may be granted if `Plan` is executed and `Losses1` and `Losses2` unify with the sanctions that can result from executing the plan which an agent is checking.

Before the agent makes the decision of intending/committing a plan, it uses Algorithms 12 and 13 and the `helpful-rule` to find the most profitable plan. We define `Bel` as the belief base that represents the agent's knowledge about the society along with the society's norms represented in EC. `TempBel` is a copy of `Bel`. We also define $\Omega$ to refer to EC1′, EC3′, Ax1, Ax2, Ax3, Ax4, Ax5, helpful-rule, `TempBel` and `Bel`.

The input of Algorithm 12 is $\Pi$, the set of applicable plans. In Line 2, we define `UtilSet` as an empty set. This set will be used to store the plans associated with their utility. In Line 4, each applicable plan is sent to Algorithm 13 and the result is

stored in `TempBel`.

In Algorithm 13, Line 2, the agent finds the sequence of actions of a plan using act($\pi$) function, which is a function that returns the sequence of actions of a given plan $\pi$. The agent asserts the actions of plan $\pi$ using the predicate `happens`, starting from time `T`, which represents the current time. Note that the added actions have not occurred yet. By adding these actions to the agent's `TempBel` belief base, the agent pretends that he has performed the actions in order to detect if the current plan $\pi$ is helpful (has positive utility) or not. Plan $\pi$ is helpful if the predicate `holdsAt(help(`$\pi$`),T)` is deduced from `TempBel` belief base; hence, the total value of all rewards is more than the total value of all losses.

In line 6 of Algorithm 12, after `holdsAt(help(`$\pi$`),T)` is deduced, the variables `Points`, `Wins`, `Losses1` and `Losses2` are unified with a set of values based on `.findall()` predicate which finds all norms violations and fulfillment (see `helpful-rule` above). In lines 7 and 8, the plan's utilities are computed and saved in the `UtilSet`. In line 11, function `findMaxUti()` returns the plan of maximum utility. This plan is then ready for execution by adding it to the agent's intentions.

When the intended plan is executed, the `happens` predicate for each executed action is added to the agent's belief base `Bel`. If the last action is asserted at time `Tn` then predicate `happens(*,Tn+1)` is asserted. The purpose of this addition is to terminate the fluents `help`, `fPun`, `oPun` and `oRew` after `Tn`. In this manner, we prevent an agent from receiving an additional sanction or reward for a past norm's violation or fulfillment that was already sanctioned or rewarded respectively. However, our mechanism can discover the violation/fulfillment that results from combining the current plan with past executed plan by modifying the axioms Ax2 through Ax5 (as

we stated at the beginning of this section).

**Algorithm 12** Find the best plan

```
1: function FindBestPl(Π)
2:     UtilSet ← {}
3:     for all π ∈ Π do
4:         TempBel ← InsertAc(π,Bel)
5:         T ← current time
6:         if Ω ⊨ holdsAt(help(π),T) then
7:             utility(π)← Points + sum(Wins) - sum(Losses1) - sum(Losses2)
8:             UtilSet ← UtilSet ∪ utility(π)
9:         end if
10:    end for
11:    BestPlan ← findMaxUti{UtilSet}
12:    return BestPlan
13: end function
```

**Algorithm 13** Add plan's actions to agent's temporary belief base

```
1: function InsertAc(π,B)
2:     X ← act(π)
       X = (A1, A2, ···, An) | Ai is an action of plan π
3:     T ← current time
4:     for i ← 1, n do
5:         B ← B ∪ happens(Ai,T)
6:         T ← T + 1
7:     end for
8:     return B
9: end function
```

## 5.4 Utilizing Permission Norms in BDI Normative Practical Reasoning

Considerable effort has been put into formalizing norms and utilizing them in agents' decision making (see López, 2003; Conte et al., 1999; Meneguzzi and Luck, 2009).

Most of this work focuses on prohibition and obligation norms (see Section 7.1.2). While these norms might be sufficient for agents with a complete knowledge of the norms in their system, we argue that these are not enough for agents with an incomplete knowledge of norms. This incomplete knowledge can originate for several reasons, including deficient norm identification techniques, changing or emerging norms, etc.

In this section, we argue that permission norms are fundamental for modeling unknown normative states, and we propose a formal representation for these norms in event calculus (EC). Using a simple mineral mining scenario implemented in Jason, a popular agent programming language, we show how to use this formal representation in agent normative practical reasoning.

### 5.4.1 Why permission norms?

A substantial amount of recent work focuses on normative practical reasoning using a variety of mechanisms. Panagiotidi and Vázquez-Salceda (2012) focus on planning based normative reasoning, in which agents form goals from norms. Criado et al. (2010) develop an agent architecture that reasons about the agent objectives based on norms. Meneguzzi et al. (2012) develop a mechanism to steer existing agent behaviour towards norm achievement while executing plans to achieve agent goals.

In these efforts, only obligation and prohibition norms are considered in normative agent decision-making, with the unstated assumption that agents are completely aware of all norms (Kollingbaum, 2005; Meneguzzi and Luck, 2009; Oren et al., 2011; Alechina et al., 2012). In these systems, agents check whether performing a particular

behaviour complies with obligations or violate prohibitions, making compromises in order to perform norm compliant behaviours. Consequently, processing permission norms is often ignored in agents' practical reasoning. This design choice seems to stem from the adoption of the Sealing Principle: "whatever is not prohibited is permitted" (Royakkers, 1997).

The sealing principle is sound if agents have complete knowledge about the normative states of a particular system, enabling them to determine whether some action violates a norm or not 100% of the time. Such a clear-cut division of the state-space is illustrated in Figure 5.2, which depicts an agent's complete knowledge of a system's normative states in accordance with the sealing principle. In this illustration, all states are identified by agents as prohibited (F) or obliged (O) and all states that aren't prohibited or obliged are identified as permitted (P). In this case, explicit reasoning about permission norms is not required since permission norms simply represent the absence of prohibition. Royakkers (1997) refers to this kind of permission (i.e., one that is not enacted by an authority) as weak permission.

However, a different division of the state-space is possible. In an alternative system, agents can have an incomplete knowledge about normative states, and the status of actions that are not known as prohibited, obliged, or permitted, is unknown. Thus, in normative terms, world states can be either obliged, prohibited, permitted, or unknown. This division of the state space is illustrated in Figure 5.3, which depicts an agent's incomplete knowledge about a system's normative states. In the illustration, agents know some states as prohibited (F), obliged (O), or permitted (P), and the rest of the state space is unknown (U).

By adding permission norms to the reasoning mechanism, the agent will be able to

reason about preferences for behaviours that are known as permitted over behaviours that are unknown. For example, consider the situation in which an agent needs to navigate from A to B and there are two paths X and Y. If the agent identifies that taking path Y is permitted and taking X is unknown, then a rational agent should take path Y rather than X (assuming X and Y have the same cost).

Consequently, we add the idea of utilizing permission norms in the agents' practical reasoning; we present a formal representation for permission norms and integrate this into a normative reasoning mechanism that also reasons about prohibition and obligation norms. We assume that agents have a mechanism to discover norms as agents explore the state-space (see Chapter 3).

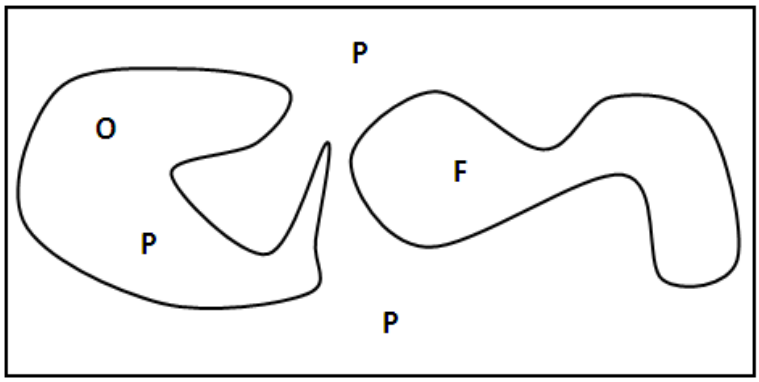


Figure 5.2: An agent's complete knowledge of the norms within a system. F represents prohibition states, O represents obliged states, and P represents permission states. P in the shape containing O refers to the implicit permission norm

### 5.4.2 Permission norm representation

In this section, we follow the norm representation presented in Section 5.2 and we expand on it by adding the permission norm to the formalization.

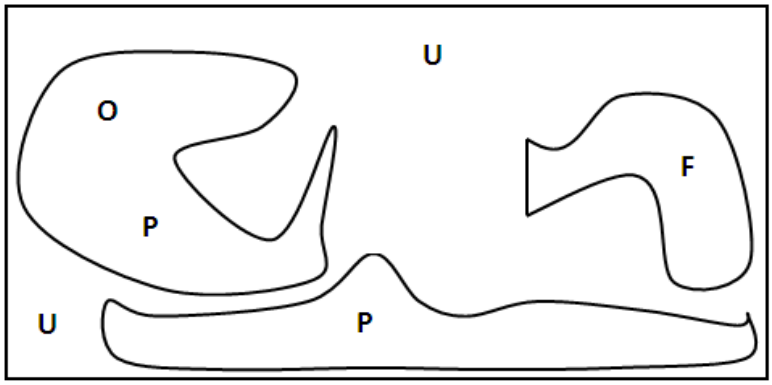


Figure 5.3: An agent's incomplete knowledge of the norms within a system. F represents prohibition states, O obliged states, P permission states, and U unknown states. P in the shape containing O refers to the implicit permission norm

In Section 5.2, three fluents are defined, one for prohibition norms (`fPun(Nid,S)`) and two for obligation norms: (`oPun(Nid,S)` and `oRew(Nid,R)`). If fluent `fPun(Nid,S)` becomes true, this means a prohibition norm has been violated. If `oPun(Nid,S)` becomes true, an obligation norm has been violated. On the other hand, if `oRew(Nid,R)` becomes true, an obligation norm has been fulfilled. In other words, these fluents work like flags that get raised if a prohibition is violated or an obligation norm is either fulfilled or violated. Such an approach is untenable for defining a fluent for permission norms. Regardless of whether the agents act according to a permission norm or not, there is no sanction or reward involved and so the fluent cannot be defined making reference to these things. Instead, of relying on sanctions or rewards to choose between plans, we want an agent to be able to select a plan based on the number of permitted actions involved. Plan `X` is then preferred over plan `Y` if `X` has more permitted actions than `Y`. For that purpose, we define the following fluent.

`pRew(Nid,1)` is a fluent which becomes true if a permitted sequence of actions has been performed, where `Nid` is the norm identification number (unique number for each permission norm). The second argument of the fluent is used to count the

number of permission norms if a plan is performed.

Informally, we define a permission norm as follows:

**Permission norm**

In a particular context, if the occurrence of a sequence of actions (or world state) is not subject to punishment, then this sequence of actions (or world state) is permitted in that context.

We use fluent `pRew(Nid,1)` in the representation of permission norms:

`initiatesAt(An,pRew(Nid,1),Tn,Tn+1):- C, happens(A1,T1)` & $\cdots$ &
`happens(An,Tn)` & `T1 <T2<` & $\cdots$ `<Tn`.

This representation contains the following parts (for an overview of our view of norms, see Section 2.1.1). `D`, the deontic type, is permission; in the left-hand side of the definition we use the `pRew` fluent which refers to performing a sequence of actions that is permitted. The second argument of the fluent `pRew` equals one in order to count the number of times a plan complies with permission norms. `C` is the norm's context. `Seq` is a sequence of actions, `A1,A2,`$\cdots$`,An`, that an agent is permitted to perform.

This representation states that, if the context `C`, entailed from the agent belief base, and the sequence of actions `A1,A2,`$\cdots$`,An` occur at time `T1,T2,`$\cdots$`,Tn` respectively, then, after time `Tn`, the fluent `pRew(Nid,1)` becomes true. We illustrate this representation using a blocks-world scenario in the example below.

**Example 5.4.1.** Suppose we have three coloured blocks, red, blue and green, and the following situation: `on(red,blue)`, `on(blue,table)` and `on(green,table)`. If

we have a permission norm that states "it is permitted to put green on red if red is not on the table", then this permission norm is represented as follows:

```
initiatesAt(on(green,red),pRew(Nid,1),T1,T2):-
¬holdsAt(on(red,table),T1) & happens(on(green,red),T1) & T1<T2.
```

## 5.5 Normative reasoning mechanism using permission norm

In order to develop our current normative reasoning mechanism, we leverage the normative reasoning mechanism presented in Section 5.3, which utilizes prohibition and obligation norms to find the best plan. We define the best plan for an agent as a plan of maximum utility for that agent (taking prohibition and obligation norms into consideration). In the context of BDI agents, the best plan is found among the applicable plans.

Our extension in this section is to utilize permission norms in order to find the safest plan among the set of best plans. If the agent finds more than one plan with the same maximum utilities, these plans are stored in set `BestSet`. We define the set of safest plans `SafestPl` as the subset of `BestSet` which contains these plans that comply with the highest number of permission norms.

We argue that using permission norms in practical reasoning within a normative system is important for at least two reasons. First, if it is the agents' duty to infer norms, the norm identification mechanism can miss some norms. Second, norms are

not fixed; they may change, emerge or vanish. Hence, presuming that "whatever is not prohibited is permitted" is not adequate since it does not account for such missing norms.

We illustrate this argument with the following scenario. Suppose that an agent wants to achieve a goal G and there are several plans for achieving G. Out of those plans the agent finds that `BestSet` has two plans, P1 and P2, of maximum utility subject to prohibition and obligation norms. Suppose that P1 has some prohibited action(s) but because of the agent's incomplete knowledge, the agent does not know that. As a result, the agent may mistakenly presume the action(s) are permitted. However, if the agent maintains the permission norms as it does the prohibition and obligation norms, then it can compare P1 and P2 to see which plan complies with the most permitted actions. Thus, it can determine which plan is the safest.

In Figure 5.4, the basic BDI interpreter overview is illustrated using white boxes and our proposed additions are presented using grey boxes. To deal with dynamic norms, the norm identification process needs to be integrated with the normative reasoning strategy in order to update an agent's belief base about repealed and emerged norms online.

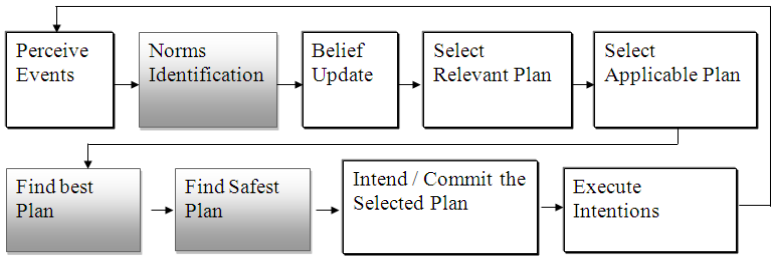


Figure 5.4: An extended BDI Reasoning processes flow

The execution of `Plan` makes fluent `help(Plan)` true if it results in more rewards than punishments (based on `helpful-rule` rule presented in Section 5.3, repeated below for convenience).

**helpful-rule:**

```
initiatesAt(_,help(Plan),T1,T2):-
.findall(V1,holdsAt(oRew(,V1),T2+1),Wins) &
.findall(V2,holdsAt(fPun(_,V2), T2), Losses1) &
.findall(V3,holdsAt(oPun(_,V3), T2), Losses2) &
goalpreference(G,Points) &
(Points + sum(Wins)- sum(Losses1) - sum(Losses2)) > 0
```

To utilize permission norms in the normative reasoning mechanism, we define `safe(Plan)` fluent which will be true if the execution of `Plan` complies with one or more permitted norms (based on the `safe-rule` below). If we have two plans with equal utility, then the plan that complies with more permitted norms is the safest. This is because the actions that are unknown might be prohibited. For each permission norm, a particular plan complies with, 1 is added to the set `Count`. Hence, the number of permission norms which a plan complies with is equal to the summation of `Count`'s elements.

**safe-rule:**

```
initiatesAt(K,safe(Plan),T1,T2):-
.findall(V1,holdsAt(pRew(_,1),T2),Count).
```

We define the following domain-independent axioms using the EC framework. Our agents implement these axioms in their normative reasoning mechanism:

- `EC1′` & `EC3′` (see Section 5.1)
- `Ax1: between(A,T1,T2) :- happens(A, T)` & `T1 <T` & `T <T2`
- `Ax2: terminatesAt(*,help(P),T1,T2):- happens(*,T1)` & `T1<T2`
- `Ax3: terminatesAt(*,safe(P),T1,T2):- happens(*,T1)` & `T1<T2`
- `Ax4: terminatesAt(*,fPun(I,S),T1,T2):- happens(*,T1)` & `T1<T2`
- `Ax5: terminatesAt(*,oPun(I,S),T1,T2):- happens(*,T1)` & `T1<T2`
- `Ax6: terminatesAt(*,oRew(I,S),T1,T2):- happens(*,T1)` & `T1<T2`
- `Ax7: terminatesAt(*,pRew(I,S),T1,T2):- happens(*,T1)` & `T1<T2`

These axioms help agents find a potential norm violation/fulfillment that can result from executing the current plan. For more details of using these axioms and the special event * please see Section 5.3.

Using the above domain-independent axioms, `helpful-rule` and `safe-rule`, the agent is able to find the set of best plans (`BestSet`) among the applicable plans, and out of the best plans the agent is able to find the safest plan.

After `helpful-rule`, the agent uses `safe-rule` to obtain the number of permission norms that a plan complies with. By using the `.findall(V,holdsAt(pRew(_-,1),_),Count)` predicate, the agent obtains all the values of `V` where fluent `pRew` is true, and adds them to the set `Count`. Note that the variable `V` is always unified with

the second argument of the fluent `pRew(_,1)`, which is always equal to one. Hence, the elements of the set `Count` are all ones and the cardinality of the set `Count` is equal to the number of permission norms that are complied with if `Plan` is executed.

**Algorithm 14** Find Safest Plan

```
1: function FindSafestPl(Π)
2:     UtilSet, BestSet, SafeSet ← {}
3:     for all π ∈ Π do
4:         TempBel ← InsertAc(π,Bel)
5:         T ← current time
6:         if Ω′ ⊨ holdsAt(help(π),T) then
7:             utility(π)← Points + sum(Wins) - sum(Losses1) - sum(Losses2)
8:             UtilSet ← UtilSet ∪ utility(π)
9:         end if
10:    end for
11:    BestSet ← BestSet ∪ findMaxSetUti{UtilSet}
12:    for all π ∈ BestSet do
13:        TempBel ← InsertAc(π,Bel)
14:        T ← current time
15:        if Ω′ ⊨ holdsAt(safe(π),T) then
16:            preference(π) ← sum(Count)
17:            SafeSet ← SafeSet ∪ preference(π)
18:        end if
19:    end for
20:    SafestPlan ← findMaxUti{SafeSet} /*max{SafeSet} returns the plan of max-
       imum preference value (the one that complies with more permission norms)*/
21:    return SafestPlan
22: end function
```

In Algorithm 14, Line 2 we define three empty sets: `UtilSet` to store a set of plans with their utilities, `BestSet` which is used to store the best plans with their utilities and `SafeSet` which stores the best plans with their number of times they comply with permission norms. As shown in Algorithm 14, Line 4 and Algorithm 13, the predicates `happens`, (see Table 2.1), are added starting from time `T`, the current time. The actions specified in predicate `happens` have not occurred yet. By adding the

plan's actions, the agent simulates that it has executed the actions in order to reason about whether the current plan $\pi$ is helpful or not. In Algorithm 14, Lines 6 and 15, $\Omega'$ refers to EC1′, EC3′, Ax1, Ax2, Ax3, Ax4, Ax5, Ax6, Ax7, helpful-rule, safe-rule and `TempBel`. In Line 6, plan $\pi$ is helpful if the predicate `holdsAt(help(`$\pi$`),T)` is deduced from `TempBel` belief base. If that is the case, the rewards outweigh losses and the plan of maximum utility is then added to the best plan set `BestSet`. The set `BestSet` will thus have the plans of maximum utilities. As a result of firing the `helpful-rule`, the variables `Points`, `Wins`, `Losses1` and `Losses2` are unified with a set of values based on `.findall()` predicate which finds all norms violations and fulfillment. In Line 11, the function `findMaxSetUti()` finds the set of plans of highest utility out of the `UtilSet` and store them in the `BestSet`.

As we see in Algorithm 14, Lines 12-21, the safest plan is found in `BestSet`. At Line 15, if the predicate `holdsAt(safe(`$\pi$`),T)` is deduced from the `TempBel` belief base, this implies that there is at least one permission norm being complied with as a result of executing plan $\pi$. In case of executing a plan, the number of times permission norms are complied with is equal to the summation of `Count` set. Plans associated with its summation of `Count` are added to the `SafeSet` set (see Algorithm 14, Line 17). In Line 20, out of the `SafeSet` set, the plan of maximum preference value is selected as the safest plan using the function `findMaxUti()`, which returns the plan of maximum preferences. The safest plan will be ready for execution by adding it to the intentions.

After choosing and performing a plan $\pi$, the `happens` predicate for each action of an executed plan $\pi$ will be added to the `Bel` belief base. Predicate `happens(*,Tn+1)` is added to the belief base after executing the last action of the chosen plan. The purpose of adding the special event * is to terminate the fluents `help, safe, fPun,`

`oPun, oRew and pRew` after `Tn+1`. This termination is important in order to prevent our agent from re-detecting a past violation. In other words, agents should not be sanctioned more than one time for the same violation. However, we do want to detect violations/fulfillments that may result from combining the current plan and the previous executed plan, which our mechanism is able to do.

## 5.6 Summary

Our extension to the classical EC allows for a formal representation of obligation and prohibition norms that can be composed of several actions. It also provides a mechanism to reason about AgentSpeak plans, taking into consideration the society's norms and the agent's past actions. For norm representation, we introduce three fluents: `fPun` to refer to prohibition norm violation, `oPun` to refer to obligation norm violation and `oRew` for obligation norm fulfillment.

Our proposed norm representation is able to represent norms that are composed of several actions along with the norm's context. Our proposed normative reasoning mechanism helps our agent to choose the most profitable plan. It takes into consideration the potential violation/fulfillment with the current plan and the potential violation/fulfillment that may result from the combination of the agent's past actions and the actions of the current plan. The most profitable plan is the plan with the highest utility. Therefore, our agent might choose anti-social plan, a plan that violates some norms but have utilities more than losses.

We present a formal representation of permission norm and integrate it into our normative reasoning mechanism based on event calculus. In addition to prohibition

and obligation norms, we design a mechanism that takes permissions into consideration to reason about the "safest" plans to execute. Such safety refers to minimizing uncertainty when an agent operates in environments with no guarantee of full knowledge of norms.

To demonstrate the operationalization, in the next chapter we describe our experiments testing the proposed mechanisms and present their results.

# Chapter 6

# Experiment results for normative reasoning

## 6.1 Experimental setup

In this chapter, we build a gold and silver mining society where gold and silver pieces are scattered in a grid-like territory along with agents who want to collect the scattered pieces into their respective ores depots (one for silver and one for gold). Once the gold and silver have been collected, the game is over and the performance of the competitor agents is compared. The competitor agent is associated with a goal and is engaged in this society. We assume that there is a set of norms which govern the society and we assume the norms are already identified and represented in the belief base.

In this society the possible actions which agents can perform are `pick(_)`, `drop(_-`

`,_)` and `moveto(_,_)`. The two competitive agents have one continuous goal, `!collect(gold)`. The importance of achieving this goal is specified by the predicate `goalpreference(collect(gold),10)`. Hence, the value of the importance of achieving the goal `!collect(gold)` is 10. In this chapter, we have two sets of experiments: one to show that our agent is able to utilize our reasoning mechanism and choose the best available behaviour (the behaviour of highest utility) in the presence of norms. In the second set of experiments we demonstrate the significance of permission norm in practical normative reasoning. We implemented our normative reasoning mechanisms in Java JDK 1.6.0 within Jason 1.3.4. The experiment was executed ten times and the average taken.

## 6.2 Experiment set 1 - Gold and silver mining society

In this scenario, the society has 20 gold ores and 20 silver ores, and five agents. One agent, called *OurAgent*, is equipped with our normative reasoning mechanism, and a different agent, who we will refer to as *OtherAgent*, randomly chooses a plan from the applicable plans. The remaining three agents do not have direct roles in our current comparison but exist in the society to reflect the fact that other agents are sharing the society. If the two agents under comparison spend a long time in the reasoning process, then the other agents can move ahead and collect all or most of the ores.

We add *OurAgent* and *OtherAgent* to this society to achieve their goals, taking into consideration the society's norms. Both agents are equipped with the the plan library and norms presented below.

**Plan Library:**

`@plan1-1`, the agent collects gold to the silver depot.

```
+!collect(gold): free ← !find(gold,X,Y); moveto(X,Y);
pick(gold); moveto(silver_depotX, silver_depotY);
drop(gold,silver_depot).
```

`@plan1-2`, the agent collects gold to the gold depot.

```
+!collect(gold): free ← !find(gold,X,Y); moveto(X,Y);
pick(gold); moveto(gold_depotX,gold_depotY); drop(gold,gold_depot).
```

`@plan1-3`, the agent collects gold and silver to their depots.

```
+!collect(gold): free ← !find(gold,X,Y); moveto(X,Y);
pick(gold); moveto(gold_depotX,gold_depotY); drop(gold,gold_depot);
!find(silver,X1,Y1); pick(silver);
moveto(silver_depotX,silver_depotY); drop(silver,silver_depot).
```

`@plan1-4`, the agent collects two gold ores to the gold depot.

```
+!collect(gold): free ← !find(gold,X,Y); moveto(X,Y);
pick(gold); moveto(gold_depotX,gold_depotY); drop(gold,gold_depot);
!find(gold,X1,Y1); moveto(X1,Y1); pick(gold);
moveto(gold_depotX,gold_depotY); drop(gold,gold_depot).
```

We suppose that the mineral mining society have the norms given below:

Prohibition Norms:

It is prohibited to drop gold in the silver depot. The sanction value is 5.

initiatesAt(drop(gold,silver_depot), fPun(1,5),T1,T2):-
happens(drop(gold,silver_depot),T1) & T1≤T2.

It is prohibited to drop silver in the gold depot. The sanction value is 15.

initiatesAt (drop(silver,gold_depot), fPun(2,15),T1,T2):-
happens(drop(silver,gold_depot),T1) & T1≤T2.

It is prohibited to carry more than one gold piece at the same time. The sanction value is 10.

initiatesAt(pick(gold), fPun(3,10),T1,T2):-
happens(pick(gold),T1) & happens(pick(gold),T2) &
T1<T2 & ¬between(drop(gold,_),T1,T2).

Obligation Norms:

It is obligatory to collect silver immediately after collecting gold. The sanction value is 20. The reward for adhering is 10.

initiatesAt(pick(gold),oPun(1,20),T1,T4):-

happens(pick(gold),T1) & happens(drop(gold,_),T2) &
happens(pick(gold),T3) & ¬between(pick(silver),T2,T3) &
T1<T2 & T2<T3 & T3≤T4.

initiatesAt(pick(gold),oRew(1,10),T1,T4):-
happens(pick(gold),T1) & happens(drop(gold,_),T2) &
happens(pick(gold),T3) & between(pick(silver),T2,T3) &
T1<T2 & T2<T3 & T3≤T4.

In Figure 6.1, the experiment results show that our proposed normative reasoning mechanism allows *OurAgent* to achieve his continuous goal as well as maximize his accumulative utilities. In contrast to *OtherAgent*, the accumulative utilities of *OurAgent* are higher because *OurAgent* uses our mechanism and always chooses the plan of the highest utility. As we see at the $x$-axis, the two agents under comparison collected a total of just 11 ores of gold and silver, out of 40. This is because the other three agents did not rely on the plan library, but just collected any piece they found. Figure 6.1 shows the result when the agent's past actions are not included in our normative reasoning mechanism. Based on the prohibition and obligation norms that *OurAgent* is aware of, `plan1-3` is the one which *OurAgent* chooses.

Figure 6.2 shows the result when the agent's history is taken into consideration. In this experiment, the agent does not look far back in its history; only one performed plan is included. As Figure 6.2 shows that as long as *OtherAgent* collects ores it loses utility because it does not have a mechanism to guide him to avoid the plans that violate norms. This experiment was repeated ten times and the average results were recorded.

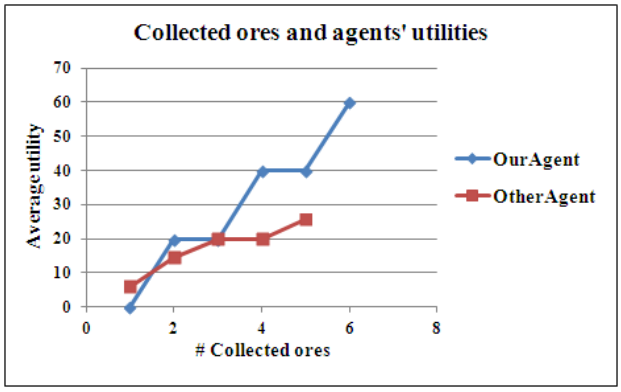


Figure 6.1: Performance comparison of *OurAgent* and *OtherAgent*.

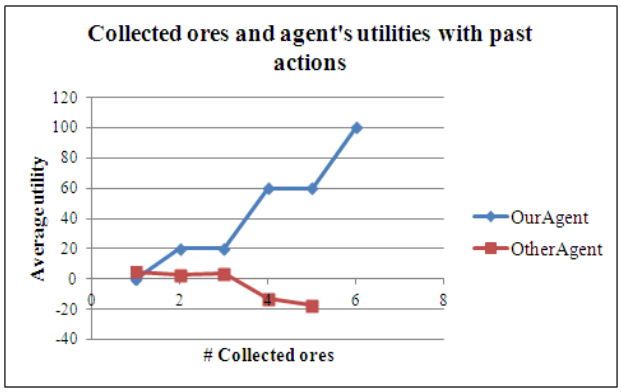


Figure 6.2: Performance comparison of *OurAgent* and *OtherAgent* (past actions included).

## 6.3 Experiment set 2 - Gold and silver mining society

In this scenario we change in the experiment setup in order to be suitable to test the significance of utilizing permission norm in agent practical normative reasoning.

In this experiments, we have three agents: a monitor agent which is able to observe other agents' actions and is also able to issue sanctions or rewards, and two other competitor agents. The two agents compete to collect 10 gold ores and 10 silver

ores. In this experiment the potential violation/fulfillment that can result from the current plan and the previously executed plan are taken into consideration.

There is a set of norms which govern this society. We assume that agents do not know all the norms. The first agent uses prohibition and obligation norms in its practical reasoning. The second agent uses prohibition, obligation and permission norms. These two agents are both aware of the same prohibition and obligation norms and are in competition with each other. The third agent is the monitor agent. Let us call the first agent the *best-agent*, the second the *best-safest-agent*, and the third the *monitor-agent*. The *best-agent* uses Algorithm 12 presented in Chapter 5 in its practical reasoning. The *best-safest-agent* uses Algorithm 14. The norms are represented using EC. The *monitor-agent* uses the axioms `Ax1` to `Ax7` to check if a violation/fulfillment occurred. The two agents have the following plans for achieving their goals (`!collect(gold)`).

`@plan1-1`, the agent collects gold to the silver depot.

```
+!collect(gold): free ← !find(gold,X,Y); moveto(X,Y);
pick(gold); moveto(silver_depotX, silver_depotY);
drop(gold,silver_depot).
```

`@plan1-2`, the agent collects gold to the gold depot.

```
+!collect(gold): free ← !find(gold,X,Y); moveto(X,Y);
pick(gold); moveto(gold_depotX,gold_depotY); drop(gold,gold_depot).
```

`@plan1-3`, the agent collects gold and silver to their depots.

```
+!collect(gold): free ← !find(gold,X,Y); moveto(X,Y);
```

```
pick(gold); moveto(gold_depotX,gold_depotY); drop(gold,gold_depot);
!find(silver,X1,Y1); pick(silver);
moveto(silver_depotX,silver_depotY); drop(silver,silver_depot).
```

`@plan1-4`, the agent collects two gold ores to the gold depot.

```
+!collect(gold):  free ← !find(gold,X,Y); moveto(X,Y);
pick(gold); moveto(gold_depotX,gold_depotY); drop(gold,gold_depot);
!find(gold,X1,Y1); moveto(X1,Y1); pick(gold);
moveto(gold_depotX,gold_depotY); drop(gold,gold_depot).
```

`@plan1-5` the agent collects gold and silver and deposits them in the gold depot.

```
+!collect(gold):  free ← !find(gold,X,Y); moveto(X,Y);
pick(gold); moveto(gold_depotX,gold_depotY); drop(gold,gold_depot);
!find(silver,X1,Y1); pick(silver); moveto(gold_depotX,gold_depotY);
drop(silver,gold_depot).
```

There are a set of prohibition, obligation and permission norms that govern this society. The prohibition and obligation norms given below are known for the three agents:

It is prohibited to drop gold in the silver depot. The sanction value is 5.

`initiatesAt(drop(gold,silverDepot),fPun(1,5),T1,T2):-`
`happens(drop(gold,silverDepot),T1)` & `T1` $\leq$ `T2`.

It is prohibited to carry more than one gold piece at a time. The sanction value is 10

initiatesAt(pick(gold),fPun(3,10),T1,T3):-
happens(pick(gold),T1) & happens(pick(gold),T2) &
¬between(drop(gold,-),T1,T2) & T1<T2 & T2≤T3.

It is obligatory to collect silver immediately after collecting gold. The sanction value is 10. The reward of adhering is 10.

initiatesAt(pick(gold),oPun(1,10),T1,T4):-
happens(pick(gold),T1) & happens(drop(gold,-),T2) &
happens(pick(gold),T3) & ¬between(pick(silver),T2,T3) &
T1<T2 & T2<T3 & T3≤T4.

initiatesAt(pick(gold),oRew(1,10),T1,T4):-
happens(pick(gold),T1) & happens(drop(gold,-),T2) &
happens(pick(gold),T3) & between(pick(silver),T2,T3) &
T1<T2 & T2<T3 & T3≤T4.

In addition to the previous norms *best-safest-agent* is aware of the following permission norms:

It is permitted to drop gold in gold depot.

initiatesAt(drop(gold,goldDepot),pRew(1,1),T1,T2):-
happens(drop(gold,goldDepot),T1) & T1≤T2.

It is permitted to drop silver in silver depot.

initiatesAt(drop(silver,silverDepot),pRew(2,1),T1,T2):-
happens(drop(silver,silverDepot),T1) & T1≤T2.

The *monitor-agent* aware of one further prohibition norm that is unknown to other agents:

It is prohibited to drop silver in the gold depot. The sanction value is 15.
`initiatesAt(drop(silver,goldDepot),fPun(1,15),T1,T2):-`
`happens(drop(silver,goldDepot),T1)` & `T1`$\leq$`T2`.

Two values for each agent was recorded: `calculated-utility`, which results from the agent's prediction in case a particular plan is chosen, and `real-utility`, which results from the execution of a particular plan. These two values could be different; for example, if an agent did not know that a particular act was prohibited, then the sanction value of performing this act could not be calculated in `calculated-utility` but it would be included in `real-utility` (the sanction value would be issued by the monitor agent).

Based on the norms and the five plans above, *best-agent* and *best-safest-agent* find the two best plans, both with utility equal to 20 (`plan1_3` and `plan1_5`). Neither agent is aware that `plan1_5` violates a prohibition that is unknown to them. Because *best-agent* has no other information to act upon, it randomly chooses between `plan1_3` and `plan1_5`, invoking a sanction from *monitor-agent* if `plan1_5` is selected. However, *best-safest-agent* selects the plan with more permission norms out of the best plans; which is `plan1_3` in this case. Thus *best-safest-agent* successfully avoids receiving a sanction that would have occurred from unknowingly violating a prohibition norm following `plan1_5`. Note that in our scenario agents do not get points for collecting silver.

The results illustrated in Figure 6.3 show that the average utility for goals achieved by *best-safest-agent* is greater than the utility of *best-agent*. This is because

*best-safest-agent* is able to integrate permission norms into its normative practical reasoning. However, *best-agent* collects more gold and silver ores than *best-safest-agent* because, while *best-safest-agent* is spending more time in the reasoning process of plan selection, *best-agent* is able to spend that time mining. Thus, *best-agent*, compared to *best-safest-agent*, presents the possibility of a higher reward (e.g., because it spends more time collecting gold and silver), but it also presents a higher risk, since it can unknowingly incur sanctions, losing an unknown amount of its reward.

The results in Figure 6.4 show that the real utility (after plan execution) of *best-agent* is less than the predicted/calculated utility (before plan execution). This is because *best-agent* does not utilize permission norms in its practical reasoning. In contrast, the real and calculated utilities were identical for *best-safest-agent*; hence, in Figure 6.4 the line for *best-safest-agent*'s calculated utility can't be seen (i.e., it is underneath the line for *best-safest-agent*'s real utility).

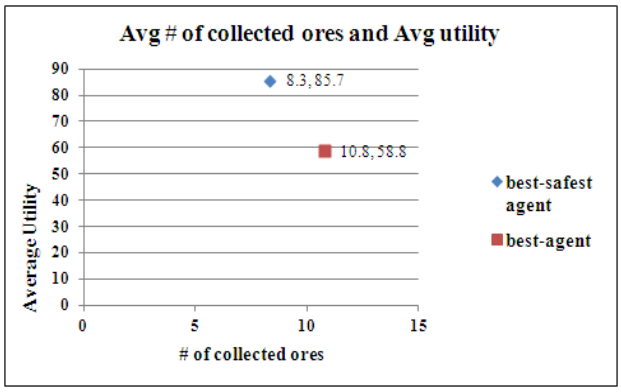


Figure 6.3: Average utility of *best-safest-agent* and *best-agent*. The average collected gold and silver ores and the ultimate utilities for *best-agent* and *best-safest-agent*.

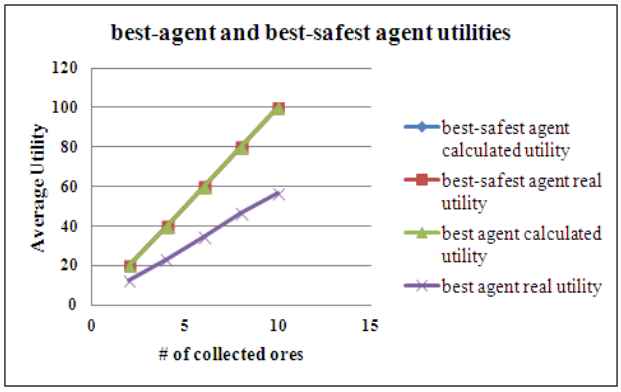


Figure 6.4: Calculated and predicted utility for *best-safest-agent* and *best-agent*.

## 6.4 Experiment discussion

To demonstrate the success of our approach, we developed a simple mineral mining scenario using the Jason BDI interpreter (Bordini et al., 2007). We used our EC extension as a formal language to represent norms and reason about the best plan that avoids prohibition and obligation norm's violations. In the first set of experiments, *OurAgent* achieved better results than *OtherAgent* since our agent was able to reason about best plan (the plan of highest utility). However, if our agent finds more than one plan with same highest utility, he needs a mechanism to prefer one over the other, hence the need for the second set of experiments where we test the significance of the permission norm.

We demonstrated empirically that when agents have incomplete knowledge about the norms of a system, then permissions have a significant role in normative practical reasoning. Using permission norm gives agents the ability to have preference over plans, i.e., plans containing actions that are known to be permitted over plans that contain actions whose normative status is unknown.

Experimental results show that using permissions in agent practical reasoning

provides agents with an extra tool to avoid norm violations, especially when agents operate in environments with no guarantee of full knowledge of norms. In spite of the fact that the throughput (i.e. the number of collected ores) of *best-safest-agent* is smaller than the throughput of *best-agent*, who does not use permissions in its reasoning, the ultimate utility of *best-safest-agent* is much higher than that of *best-agent* (see Figure 6.3). This result implies that agents who do not utilize permission norms in their practical reasoning can misbehave and violate unknown norms.

# Chapter 7

# Summary

Norms have been proposed as a means to configure and organize open multi-agent systems given their constantly dynamic structure. In this thesis, we studied the problems of norm identification, norm representation and normative practical reasoning. The norms that are used in the practical reasoning approach should be compatible with the norms that are identified by the norm identification algorithms. The importance of studying these aspects together becomes apparent when agents need to operate in a society in which norms are not given or predefined for the agents, or in a society with dynamic or changeable norms. In such systems, agents need to identify norms and then take them into account during their practical reasoning. Crucially, then, those norms identified during norm identification should be compatible with the view of norms during practical reasoning. Despite this, to the best of our knowledge, these two aspects have never been studied together. In this thesis, we shed light on the importance of studying these two aspects together and start the first steps toward achieving this by proposing norm identification approach and normative reasoning

mechanism.

For our research, we assumed that norms are changeable: new ones can be introduced and old ones repealed. We also assumed that it is the agents' duty to infer norms. Having made these assumptions, we needed to further assume that agents do not have complete knowledge about the normative states of a system. This is contrary to other earlier proposed normative practical reasoning mechanisms, which assume that agents have complete knowledge about the normative states.

Our main objective in this thesis is to establish a framework for creating a BDI agent capable of joining and functioning in a society regulated by (possibly unknown) norms, while minimizing behaviours that violate norms. To achieve this objective, we answered our first research question by proposing an agent architecture and algorithms for identifying permission and prohibition norms based on observation and communication, and utilizing permission norm in detecting repealed prohibition norms (Chapters 3 and 4).

Other approaches of norm identification depend on communicating with other agents using a direct inquiry about norms. However, open multi-agent systems can have heterogeneous agents; agents that are designed by different users and agents that can have different internal architectures. Consequently, it is possible that some of these agents are unaware of the concept of norm or they might use a different terminology for norm. This makes the direct inquiry about norms unsuitable for open multi-agent systems. To overcome this limitation, we use the FIPA-CFP and FIPA's Contract Net Interaction Protocol (FIPA, 2002b), to allow indirect inquiries regarding norms (see Alrawagfeh et al., 2011b). Our work in Chapters 3 and 4 enables agents to dynamically infer new and repealed norms while taking norms' consistency

into account.

To achieve our main objective, we answered the second research question in Chapters 5 and 6. We extend the classical event calculus (Kowalski and Sergot, 1989) to formalize a representation of norm and propose a mechanism for integrating norm into BDI agent practical reasoning. We define a rule called `helpful-rule` which is used by our mechanism to find the best plan in the agent's plan library. Our proposed mechanism increases the flexibility of normative practical reasoning by allowing the previous actions (an agent's history) to be considered when evaluating current plans during the reasoning process. Our mechanism thus enables our agent to discover the norm violation or fulfillment which can result from a combination of the current plan actions and the previous executed actions. Also, we proposed a formalization of permission norms and integrate it into BDI agent practical reasoning. We show how permission norms are useful in open multi-agent systems if agents do not have complete knowledge about the system's normative states.

In dynamic systems, being able to use permission norms is a significant tool during normative practical reasoning process, where agents can prefer behaviours that are known as permitted over behaviours that are not known to be permitted, prohibited or obliged. Thus, we utilized permission norms in both norm identification and normative practical reasoning.

Our work can be applied to several areas including social robotics, life style and heart-attack prediction, and offshore petroleum safety:

- Researchers in social robotics aim to create autonomous robots that interact with humans or other autonomous robots by taking social behaviours into con-

sideration. Our work can be integrated into a social robot so it can discover the norms that govern a particular society and adapt its behaviours accordingly.

- The problem of heart-attack predication can be modeled using normative multi-agent systems. By doing so, our agent plays the role of recommending a particular life style for a patient. The different life styles are represented as plans. Each human habit (smoking, drinking coffee, consuming sugar, playing sports, etc.) that may have an effect on the heart is represented as a norm, and the patient's medical history is represented as beliefs. Based on a repository of human habits that may cause heart attack, the agent identifies the habits of an individual that may cause him/her to have a heart attack and recommend a life style which has the least negative effect on the heart.

- In offshore petroleum safety, an agent might play the role of an offshore-petroleum platform's manager assistant. For example, when employees need to leave the platform, our agent advises the manager on which plan is safer to use, the helicopter or the boat. Based on several factors (such as temperature, wind speed and direction, fog, etc.) and using our normative reasoning mechanism, the agent suggests the best plan to transport the employees. Also the agent may be trained on the offshore platform to discover the employees' behaviours during disasters in order to identify the actions that can delay a potential evacuation and to recommend evacuation plans to the manager in case of a real disaster.

In general our work can be applied to any system like these above where agents can have several alternative behaviours that lead to the same outcome, and at the same time there are actions whose effects vary depending on the situation. Note that using predefined constraints in such systems is not helpful because of their dynamic

nature.

## 7.1 Related work

### 7.1.1 Norm identification

Taking into consideration that the norms in a society are subject to change or might even disappear, agents need a mechanism to infer changes in norm status. Agents may also need to employ a similar mechanism when they join a new society in order to infer the norms of the joined society. The process by which agents discover unknown norms is called norm identification. There are several different approaches to norm identification. We focus on the approaches by Savarimuthu (2011), Andrighetto et al. (2010), Mahmoud et al. (2012b), Mahmoud et al. (2012a), Oren and Meneguzzi (2013) and Savarimuthu et al. (2013).

Savarimuthu (2011) introduces the idea of identifying prohibition and obligation norms by observing both regular events and sanctions (e.g., special events). According to their model, the occurrence of a sanction event indicates the violation of a prohibition norm. Using data mining algorithms, the researchers identify candidate prohibitions, that is, sequences of actions that possibly cause sanctions. Candidate norms are then sent to a verification component to check whether candidate prohibition norms are actual norms. Experimentally, Savarimuthu et al. demonstrate the validity of their work by showing that their agents have the ability to identify new norms.

As for changeable or repealed norms, Savarimuthu et al.'s model revokes or

deletes a prohibition norm if it has not been detected by the algorithm for a certain period of time. In contrast, their model identifies a prohibition norm once it has been violated several times. Problematically, then, in cases where a particular prohibition norm has not been violated for quite some time, the norm might be revoked by mistake. For example, suppose that in a traffic situation, an agent A is aware of a norm which states that running a red light is prohibited. In Savarimuthu et al.'s model, if no agents run a red light during the period of observation (i.e., there is no norm violation), agent A will wrongly delete that norm from its belief base.

An alternative proposal is offered by Andrighetto et al. (2010), who investigate a mechanism for new norm identification in an environment simulating complex social systems. They propose a cognitive agent architecture based on mental representations which allows norms to influence the behaviour of autonomous agents. Their work is conditioned on an agent's ability to recognize an observed or communicated social input. When a social input arrives, the agent checks to see if it contains a deontic statement (such as, "You must answer when asked") or a normative valuation (such as, "Not answering when asked is impolite"). From this input, a candidate belief (such as, "One must answer when asked") is generated and temporarily stored as a candidate normative belief. At this point, several investigative processes should be executed on the social input to evaluate whether it is a norm or not. Such processes, however, have not yet been implemented in Andrighetto et al.'s work.

Mahmoud and colleagues (2012b; 2012a) propose a norm mining approach based on interaction. Their concept of norms is very general, as they do not deal explicitly with permission, prohibition or obligation norms. They assume that a visitor agent joins a society that has a facilitator agent. The latter maintains a log file of historical records of actions performed by agents. If the visitor agent is not able to access

the log file, then it communicates both with other agents and with the facilitator. Alternatively, the visitor agent can make direct observations if the log file and communications are not available. The facilitator then stores the visitors' interactions to use as an additional source for norm identification. Using the collected data, visitor agents then apply data mining algorithms to identify potential norms.

An additional approach to norm identification, by Oren and Meneguzzi (2013), is based on plan recognition. In their model, norm identification involves a plan recognizer and a planner. The recognizer agent observes other agents' actions, and then identifies other agents' goals. Subsequently, the planner generates candidate plans for these goals. The generated plans are then compared with recognized plans and the avoided or repeatedly visited actions (or states) are identified as norms. Oren and Meneguzzi also assume that a society has common domain knowledge in the form of a plan library shared among agents. This assumption narrows the applicable domains for their approach.

### 7.1.2 Normative practical reasoning

In this sub-section we discuss the literature dealing with the impact of norms on autonomous agent reasoning, reviewing several proposals for how norms might be considered when agents select and execute plans.

Kollingbaum (2005) studies the problem of norm inconsistency that can result when agents adopt new norms. He proposes a normative agent architecture and uses a programming language in his investigation of normative practical reasoning. Kollingbaum's proposed architecture is more concerned with fulfilling specific norms

than goals, and is driven by norms instead of mental states (as in traditional BDI agents).

Another approach of utilizing norms in agent practical reasoning is offered by Meneguzzi and Luck (2009). Their approach expands the functionality of BDI-based agents to allow them to change their behaviour in response to newly accepted norms. In their work, new plans are created in order to comply with obligation norms; conversely, plans are suppressed when they violate a prohibition norm (i.e., when they include at least one action that violates a norm).

We recognize that it is potentially beneficial for an agent to execute a particular plan even when the plan has an action that violates a norm. For example, a plan which violates a norm while achieving a goal and fulfilling an obligation may have rewards that offset the punishment potentially resulting from the norm's violation. Hence, our approach takes this point into consideration.

Oren and colleagues (2011) propose a technique for considering norms when describing how to execute a plan. They define norms as constraints on the values of variables in descriptions of actions. Their constraint-based norms describe the manner in which an action should be executed. Our work here differs from their work in the conception and the definition of norms.

An alternative approach is offered by Alechina et al. (2012), who further develop the BDI programming language 2APL (Dastani, 2008) to support normative concepts, including prohibition, obligation, sanction, duration and deadline. An exogenous organization sends the prohibition and obligation norms to the agent. This means that agents in Alechina et al.'s (2012) model do not have to detect norms. Similar to the model of Criado et al. (2011), obligation norms are stored in the agents' belief

base as normative goals. These goals are achieved based on priorities that are placed on sanctioning violations. Prohibition norms are stored in the agent event base. They define deadlines for executing plans and schedule feasible plans based on those deadlines. Feasible plans are checked against norms to see if they include an action that violates a prohibition norm of a priority greater than or equal to the priority of the plan. If so, the plan is not selected. In contrast, a plan in the method of Meneguzzi and Luck's (2009) is not selected if it violates even one norm, regardless of the sanction value. Alechina et al. provide an alternate approach in their model by taking the value of the sanction into account; if the violations' sanction is less than the goal preference, the plan is selected. However, their model does not take into account the accumulated sanction values which are expected to result from executing a plan.

Balke and collegues (2012) propose a methodology for normative reasoning at run-time. They develop a design-time and run-time institution (normative system) in the context of BDI agents. They assume that agents can query a special agent, InstitutionKeeper, about the normative states. Their agent reasons as follows: the agent receives percepts from the environment as well as normative percepts from InstitutionKeeper. InstitutionKeeper stores the normative state and handles the realization of all institutional norms. Before an agent executes an action, it asks InstitutionKeeper whether the action is forbidden. Thus, Balke et al.'s agents are dependent on a special agent in the institution to help their agent with its normative reasoning. We do not adopt such an approach in our model.

Meneguzzi et al. (2012) propose v-BDI as an extension to the BDI architecture to enable normative reasoning as well as selecting and customizing plans to ensure norm compliance. The normative process is applied on the entire plan library. Note,

however, that the efforts of such a process will be wasted if a plan is irrelevant or inapplicable. In contrast, we propose that the agent applies normative reasoning only to applicable plans (i.e., the plan that has context that is logically consequent from the agent's current belief base).

Artikis et al. (2005) use event calculus (see Section 2.4) to represent norms, restricting their attention to norms of obligation and permission. They assume that any action that is not permitted is prohibited. In contrast, Fornara and Colombetti (2009) suppose that any action that is neither prohibited nor obliged is permitted. To be closer to real-world scenarios, in this thesis, we assume that any action that is not prohibited nor obliged is unknown.

The work in Artikis et al. (2005) and Meneguzzi et al. (2012) do not differentiate punishment severity, but instead count the number of times a violation occurs. Problematically, any decision-making based on such a count will not result in a precise decision. This is because the punishment for one particular violation may be more severe than other punishments. For example the punishment for murder is more severe than the punishment for shouting in a library or not returning a book.

In this thesis, we extended the classical event calculus to propose a norm representation method that is powerful enough to represent more complex norms than those identified in Alrawagfeh et al. (2011c). Because of this, we are able to build on Alrawagfeh and colleagues' (2011a; 2011c) approach to make it possible to add information such as the world state, the agent's role, and more detailed context to representations.

To the best of our knowledge, previous normative reasoning strategies ignore permission norms and do not take into consideration past agents actions as we do

here in this thesis. Also, we are not aware of work in norm representation that explicitly discusses the flexibility of representing norms that are composed of several actions.

## 7.2 Limitations and future work

This thesis makes significant contributions to different and complex areas of normative multi-agent systems, in particular, norm identification, representation and practical reasoning. To do this, we made certain assumptions in order to focus on the issues being addressed:

- The events recognizer component is able to distinguish between regular and special events (sanctions). As future work, our agent might be designed to observe an event resulting in the loss of money, time, resources, reputation or respectfulness, etc., and interpret this event as a special event (sanction).

- Norms can be composed of several events. In our experiments, we assumed that the queue size (where the observed events are stored) is limited to 8 events. If we deal with a society in which norms are composed of more than 7 events, then the queue size should be increased. The queue size should be greater than the number of events that can form the norm.

- The special event should not be issued later than the violation event(s). In other words, the event(s) which cause the violation and the sanction event should appear in the queue at the same time.

- Our agent does not have complete knowledge of the joined system's normative

states. It is the agent's duty to detect the norms of the joined society.

While we have provided strong and valuable contributions to the development of norm identification and normative practical reasoning, there are still a number of avenues for future work:

1. Expand our agent architecture by creating an automated norms representation component. This component receives its inputs from the norm identification algorithms and stores the identified norms in the agent belief base represented by event calculus.

2. Develop logical rules to infer obligation norms based on prohibition and permission norms.

3. Enhance our agent's flexibility in reasoning about best plans by giving it the ability to exchange some plan's actions, which may violate norms, with other actions that have the same outcome but do not violate norms.

4. Study the runtime efficiency of our practical normative reasoning mechanism. It would also be interesting to compare our *best-safest-agent* with other BDI norm aware agents in the literature.

5. Find a method for agents to avoid the sanction of norm violation. This would better represent daily life, in which a violation can be mitigated and a sanction avoided by performing an extra act (e.g., giving an apology).

# References

Aldewereld, H., Dignum, F., García-Camino, A., Noriega, P., Rodríguez-Aguilar, J. A., and Sierra, C. (2007). Operationalisation of norms for electronic institutions. In Noriega, P., Vázquez-Salceda, J., Boella, G., Boissier, O., Dignum, V., Fornara, N., and Matson, E., editors, *Coordination, Organizations, Institutions, and Norms in Agent Systems II*, volume 4386 of *Lecture Notes in Computer Science*, pages 163–176, Berlin, Germany. Springer.

Alechina, N., Dastani, M., and Logan, B. (2012). Programming norm-aware agents. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems*, volume 2, pages 1057–1064, Richland, South Carolina. International Foundation for Autonomous Agents and Multiagent Systems.

Alrawagfeh, W. (2013). Norm representation and reasoning: A formalization in event calculus. In Boella, G., Elkind, E., Savarimuthu, B., Dignum, F., and Purvis, M., editors, *PRIMA 2013: Principles and Practice of Multi-Agent Systems*, volume 8291 of *Lecture Notes in Computer Science*, pages 5–20, Berlin, Germany. Springer.

Alrawagfeh, W., Brown, E., and Mata-Mantero, M. (2011a). Identifying norms of behaviour in multi-agent societies. In *the 7th Conference of the European Social Simulation Association (ESSA)*. Montpelier, France.

Alrawagfeh, W., Brown, E., and Mata-Mantero, M. (2011b). Identifying norms of behaviour in open multi-agent societies. In *Proceedings of the 2011 Workshop on Agent-Directed Simulation*, pages 13–20, Boston, USA. Society for Computer Simulation International.

Alrawagfeh, W., Brown, E., and Mata-Montero, M. (2011c). Norms of behaviour and their identification and verification in open multi-agent societies. *International Journal of Agent Technologies and Systems (IJATS)*, (3):1–16.

Alrawagfeh, W. and Meneguzzi, F. (2015). Utilizing permission norms in BDI practical normative reasoning. In *Proceedings of the 16th International Workshop on Coordination, Organizations, Institutions, and Norms*, Lecture Notes in Computer Science, Berlin, Germany. Springer.

Anderson, A. R. (1958). A reduction of deontic logic to alethic modal logic. *Mind*, 67(265):100–103.

Andrighetto, G., Campennί, M., Cecconi, F., and Conte, R. (2008). How agents find out norms: A simulation based model of norm innovation. In *Proceedings of the third International Workshop on Normative Multiagent Systems*, pages 16–30.

Andrighetto, G., Campennì, M., Cecconi, F., and Conte, R. (2010). The complex loop of norm emergence: A simulation model. In Takadama, K., Cioffi-Revilla, C., and Deffuant, G., editors, *Simulating Interacting Agents and Social Phenomena*, volume 7, pages 19–35, Berlin, Germany. Springer.

Artikis, A., Kamara, L., Pitt, J., and Sergot, M. (2005). A protocol for resource sharing in norm-governed ad hoc networks. In Leite, J., Omicini, A., Torroni, P., and Yolum, p., editors, *Declarative Agent Languages and Technologies II*, volume 3476 of *Lecture Notes in Computer Science*, pages 221–238, Berlin, Germany. Springer.

Balke, T., De Vos, M., and Padget, J. (2012). Normative run-time reasoning for institutionally-situated BDI agents. In Cranefield, S., van Riemsdijk, M., Vázquez-Salceda, J., and Noriega, P., editors, *Coordination, Organizations, Institutions, and Norms in Agent System VII*, volume 7254 of *Lecture Notes in Computer Science*, pages 129–148, Berlin, Germany. Springer.

Bellifemine, F. L., Caire, G., and Greenwood, D. (2007). *Developing multi-agent systems with JADE*. Wiley.

Boella, G., Pigozzi, G., and van der Torre, L. (2009). Five guidelines for normative multiagent systems. In *Proceedings of the twenty-second annual Conference on Legal Knowledge and Information Systems*, pages 21–30, Amsterdam, Netherlands. IOS Press.

Boella, G. and Van Der Torre, L. (2007a). A game-theoretic approach to normative multi-agent systems. In Boella, G., van der Torre, L., and Verhagen, H., editors, *Normative Multi-agent Systems*, number 07122 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

Boella, G. and Van Der Torre, L. (2007b). Norm negotiation in multiagent systems. *International Journal of Cooperative Information Systems*, 16(01):97–122.

Boella, G., van der Torre, L., and Verhagen, H. (2006). Introduction to normative multiagent systems. *Computational and Mathematical Organization Theory*, 12(2-3):71–79.

Boella, G., van der 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.

Boman, M. (1999). Norms in artificial decision making. *Artificial Intelligence and Law*, 7(1):17–35.

Bordini, R. H. and Hübner, J. F. (2006). BDI agent programming in AgentSpeak using Jason. In Toni, F. and Torroni, P., editors, *Computational Logic in Multi-Agent Systems*, volume 3900 of *Lecture Notes in Computer Science*, pages 143–164, Berlin, Germany. Springer.

Bordini, R. H., Hübner, J. F., and Wooldridge, M. (2007). *Programming multi-agent systems in AgentSpeak using Jason*. Wiley.

Bratman, M. (1987). *Intention, plans, and practical reason*. Harvard University Press, Cambridge, Massachusetts.

Bratman, M. E., Israel, D. J., and Pollack, M. E. (1988). Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4(3):349–355.

Braubach, L., Lamersdorf, W., and Pokahr, A. (2003). Jadex: Implementing a BDI-infrastructure for jade agents. *EXP – in search of innovation*, 3(3):76–85.

Campenní, M., Andrighetto, G., Cecconi, F., and Conte, R. (2009). Normal = Normative? The role of intelligent agents in norm innovation. *Mind and Society*, 8(2):153–172.

Castelfranchi, C. (2003). Formalising the informal?: Dynamic social order, bottom-up social control, and spontaneous normative relations. *Journal of Applied Logic*, 1(1):47–92.

Cohen, P. R. and Levesque, H. J. (1990). Intention is choice with commitment. *Artificial intelligence*, 42(2):213–261.

Conte, R. and Castelfranchi, C. (1995). Understanding the effects of norms in social groups through simulation. In Gilbert, I. N. and Rosaria Conte, e., editors, *Artificial societies: the computer simulation of social life*, pages 252–267, London, UK. UCL Press.

Conte, R., Castelfranchi, C., and Dignum, F. (1999). Autonomous norm acceptance. In Müller, J., Rao, A. S., and Singh, M. P., editors, *Intelligent Agents V: Agents Theories, Architectures, and Languages*, volume 1555 of *Lecture Notes in Computer Science*, pages 99–112, Berlin, Germany. Springer.

Criado, N., Argente, E., and Botti, V. (2011). Rational strategies for norm compliance in the n-BDI proposal. In De Vos, M., Fornara, N., Pitt, J., and Vouros, G., editors, *Coordination, Organizations, Institutions, and Norms in Agent Systems VI*, volume 6541 of *Lecture Notes in Computer Science*, pages 1–20, Berlin, Germany. Springer.

Criado, N., Argente, E., Noriega, P., and Botti, V. J. (2010). Towards a normative BDI architecture for norm compliance. In Fornara, N. and Vouros, G., editors, *11th International Workshop on Coordination, Organization, Institutions and Norms in Multi-Agent Systems*, pages 65–81, Berlin, Germany. Springer.

Dastani, M. (2008). 2APL: A practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248.

Dellarocas, C. and Klein, M. (2001). Contractual agent societies. In Conte, R. and Dellarocas, C., editors, *Social Order in Multiagent Systems*, volume 2 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 113–133, Berlin, Germany. Springer.

Dignum, F. (1999). Autonomous agents with norms. *Artificial Intelligence and Law*, 7(1):69–79.

Dignum, V., Vzquez-Salceda, J., and Dignum, F. (2005). OMNI: Introducing social structure, norms and ontologies into agent organizations. In Bordini, R., Dastani, M., Dix, J., and El Fallah Seghrouchni, A., editors, *Programming Multi-Agent Systems*, volume 3346 of *Lecture Notes in Computer Science*, pages 181–198, Berlin, Germany. Springer.

d'Inverno, M., Kinny, D., Luck, M., and Wooldridge, M. (1998). A formal specification of dMARS. In Singh, M. P., Rao, A. S., and Wooldridge, M., editors, *Agent Theories, Architectures, and Languages*, volume 1365 of *Lecture Notes in Computer Science*, pages 155–176, Berlin, Germany. Springer.

Esteva, M., Rodríguez-Aguilar, J.-A., Sierra, C., Garcia, P., and Arcos, J. L. (2001). On the formal specification of electronic institutions. In Dignum, F. and Sierra, C., editors, *Agent Mediated Electronic Commerce*, volume 1991 of *Lecture Notes in Computer Science*, pages 126–147, Berlin, Germany. Springer.

FIPA (2002a). FIPA ACL message structure specification. Document number SC00061G. http://www.fipa.org/specs/fipa00061/ (visited on 2015-02-06).

FIPA (2002b). FIPA contract net interaction protocol specification. Document number SC00029H. http://www.fipa.org/specs/fipa00029/ (visited on 2015-02-06).

FIPA (2002c). FIPA SL content language specification. Document number SC00008I. http://www.fipa.org/specs/fipa00008/ (visited on 2015-02-06).

FIPA (2004). FIPA Agent Management Specification. Document number SC00023K. http://www.fipa.org/specs/fipa00023/ (visited on 2015-02-06).

Fornara, N. and Colombetti, M. (2009). Specifying artificial institutions in the event

calculus. In *Handbook of Research on Multi-Agent Systems: Semantics and Dynamics of Organizational Models*, pages 335–366. IGI Global, Pennsylvania, USA.

Grossi, D. (2007). *Designing Invisible Handcuffs: Formal Investigations in Institutions and Organizations for Multi-agent Systems*. PhD thesis, Utrecht University.

Grossi, D., Aldewereld, H., and Dignum, F. (2007). Ubi Lex, Ibi Poena: Designing norm enforcement in E-Institutions. In Noriega, P., Vázquez-Salceda, J., Boella, G., Boissier, O., Dignum, V., Fornara, N., and Matson, E., editors, *Coordination, Organizations, Institutions, and Norms in Agent Systems II*, volume 4386 of *Lecture Notes in Computer Science*, pages 101–114, Berlin, Germany. Springer.

Hashmi, M., Governatori, G., and Wynn, M. (2014). Modeling obligations with event-calculus. In Bikakis, A., Fodor, P., and Roman, D., editors, *Rules on the Web. From Theory to Applications*, volume 8620 of *Lecture Notes in Computer Science*, pages 296–310, Berlin, Germany. Springer.

Hayzelden, A., Bigham, J., Wooldridge, M., and Cuthbert, L. (1999). Future communication networks using software agents. In Hayzelden, A. and Bigham, J., editors, *Software Agents for Future Communication Systems*, pages 1–57, Berlin, Germany. Springer.

Hollander, C. D. and Wu, A. S. (2011). The current state of normative agent-based systems. *Journal of Artificial Societies and Social Simulation*, 14(2):6. [Online]. Available: http://jasss.soc.surrey.ac.uk/14/2/6.html.

Horn, A. (1951). On sentences which are true of direct unions of algebras. *The Journal of Symbolic Logic*, 16:14–21.

Huynh, T. D., Jennings, N. R., and Shadbolt, N. R. (2006). An integrated trust and

reputation model for open multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 13(2):119–154.

Jones, A. J. I. and Sergot, M. (1993). On the characterisation of law and computer systems: the normative systems perspective. In Meyer, J.-J. and Wieringa, R., editors, *Deontic Logic in Computer Science: Normative System Specification*, pages 275–307. Wiley.

Koeppen, J. and Lopez-Sanchez, M. (2010). Generating new regulations by learning from experience. In *Proceedings of 9th workshop on Coordination, Organization, Institutions and Norms in Multi-agent Systems*, pages 72–79.

Kollingbaum, M. (2005). *Norm-governed Practical Reasoning Agents*. PhD thesis, University of Aberdeen.

Kowalski, R. and Sergot, M. (1989). A logic-based calculus of events. In Schmidt, J. and Thanos, C., editors, *Foundations of Knowledge Base Management*, Topics in Information Systems, pages 23–55, Berlin, Germany. Springer.

López, F. L. Y. (2003). *Social Power and Norms: Impact on Agent Behaviour*. PhD thesis, University of Southampton.

Mahmoud, M. A., Ahmad, M. S., Ahmad, A., Yusoff, M. Z. M., and Mustapha, A. (2012a). A norms mining approach to norms detection in multi-agent systems. In *International Conference on Computer and Information Sciences*, volume 1, pages 458–463, Los Alamitos, California. IEEE press.

Mahmoud, M. A., Ahmad, M. S., Ahmad, A., Yusoff, M. Z. M., and Mustapha, A. (2012b). The semantics of norms mining in multi-agent systems. In *Computational*

*Collective Intelligence. Technologies and Applications*, volume 7653 of *Lecture Notes in Computer Science*, pages 425–435, Berlin, Germany. Springer.

Meneguzzi, F. and Luck, M. (2009). Norm-based behaviour modification in BDI agents. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems*, volume 1, pages 177–184, Richland, South Carolina. International Foundation for Autonomous Agents and Multiagent Systems.

Meneguzzi, F., Vasconcelos, W., Oren, N., and Luck, M. (2012). Nu-BDI: Norm-aware BDI agents. In *The 10th European Workshop on Multi-Agent Systems (EUMAS)*, London, UK.

Miller, R. and Shanahan, M. (1999). The event calculus in classical logic-alternative axiomatizations. *Linkoping Electronic Articles in Computer and Information Science*, 4(16):1–27.

Oren, N. and Meneguzzi, F. (2013). Norm identification through plan recognition. In *Coordination, Organization, Institutions and Norms in Agent Systems at AAMAS13*.

Oren, N., Vasconcelos, W., Meneguzzi, F., and Luck, M. (2011). Acting on norm constrained plans. In Leite, J., Torroni, P., Ågotnes, T., Boella, G., and van der Torre, L., editors, *Computational Logic in Multi-Agent Systems*, volume 6814 of *Lecture Notes in Computer Science*, pages 347–363, Berlin, Germany. Springer.

Ostrom, E. (2014). Collective action and the evolution of social norms. *Journal of Natural Resources Policy Research*, 6(4):235–252.

Panagiotidi, S. and Vázquez-Salceda, J. (2012). Towards practical normative agents: A framework and an implementation for norm-aware planning. In Cranefield, S.,

van Riemsdijk, M., Vázquez-Salceda, J., and Noriega, P., editors, *Coordination, Organizations, Institutions, and Norms in Agent System VII*, volume 7254 of *Lecture Notes in Computer Science*, pages 93–109, Berlin, Germany. Springer.

Ramchurn, S. D., Huynh, D., and Jennings, N. R. (2004). Trust in multi-agent systems. *Knowledge Engineering Review*, 19(1):1–25.

Rao, A. (1996). AgentSpeak(L): BDI agents speak out in a logical computable language. In Van de Velde, W. and Perram, J., editors, *Agents Breaking Away*, volume 1038 of *Lecture Notes in Computer Science*, pages 42–55, Berlin, Germany. Springer.

Rao, A. S. and Georgeff, M. P. (1995). BDI agents: From theory to practice. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 312–319, Menlo Park, California. AAAI Press.

Royakkers, L. M. M. (1997). Giving permission implies giving choice. In *8th International Workshop on Database and Expert Systems Applications*, pages 198–203, Los Alamitos, California. IEEE press.

Savarimuthu, B., Padget, J., and Purvis, M. (2013). Social norm recommendation for virtual agent societies. In Boella, G., Elkind, E., Savarimuthu, B., Dignum, F., and Purvis, M., editors, *PRIMA 2013: Principles and Practice of Multi-Agent Systems*, volume 8291 of *Lecture Notes in Computer Science*, pages 308–323, Berlin, Germany. Springer.

Savarimuthu, B. T. R. (2011). *Mechanisms for Norm Emergence and Norm Identification in Multi-agent Societies*. PhD thesis, University of Otago.

Savarimuthu, B. T. R. and Cranefield, S. (2009). A categorization of simulation

works on norms. 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.

Savarimuthu, B. T. R., Cranefield, S., Purvis, M. A., and Purvis, M. K. (2010a). Internal agent architecture for norm identification. In Padget, J., Artikis, A., Vasconcelos, W., Stathis, K., da Silva, V., Matson, E., and Polleres, A., editors, *Coordination, Organizations, Institutions and Norms in Agent Systems V*, volume 6069 of *Lecture Notes in Computer Science*, pages 241–256, Berlin, Germany. Springer.

Savarimuthu, B. T. R., Cranefield, S., Purvis, M. A., and Purvis, M. K. (2010b). Norm identification in multi-agent societies. University of Otago. Available at https://ourarchive.otago.ac.nz/handle/10523/1031.

Savarimuthu, B. T. R., Cranefield, S., Purvis, M. A., and Purvis, M. K. (2011). Identifying conditional norms in multi-agent societies. In De Vos, M., Fornara, N., Pitt, J., and Vouros, G., editors, *Coordination, Organizations, Institutions, and Norms in Agent Systems VI*, volume 6541 of *Lecture Notes in Computer Science*, pages 285–302, Berlin, Germany. Springer.

Shanahan, M. (1999). The event calculus explained. In Wooldridge, M. and Veloso, M., editors, *Artificial Intelligence Today*, volume 1600 of *Lecture Notes in Computer Science*, pages 409–430, Berlin, Germany. Springer.

Shoham, Y. and Tennenholtz, M. (1995). On social laws for artificial agent societies: Off-line design. *Artificial Intelligence*, 73(1-2):231–252.

Soeteman, A. (2001). *Pluralism and law*, volume 1. Springer Science and Business Media, Berlin, Germany.

Stone, P. and Veloso, M. (2000). Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots*, 8(3):345–383.

Verhagen, H. J. (2000). *Norm Autonomous Agents*. PhD thesis, The Royal Institute of Technology and Stockholm University.

von Wright, G. H. (1968). *An Essay in Deontic Logic and the General Theory of Action*. North-Holland Publishing Company, Amsterdam, Netherlands.

Wooldridge, M. (2002). *An Introduction to Multiagent Systems*. Wiley.

Wooldridge, M. and Jennings, N. R. (1994). Agent theories, architectures and languages: A survey. In Wooldridge, M. and Jennings, N. R., editors, *Intelligent agents*, Lecture Notes in Computer Science, pages 1–39, Berlin, Germany. Springer.