{

An Agent-based Approach to the Governance of Complex Cyber-Infrastructures

}

Mostafa Mohajeri Parizi

An Agent-based Approach to the Governance of Complex Cyber-Infrastructures

Mostafa Mohajeri Parizi

This work has been done as part of the Dutch Research project *Data Logistics for Logistics Data* (DL4LD), supported by the Dutch Organisation for Scientific Research (NWO), the Dutch Institute for Advanced Logistics TKI Dinalog (http://www.dinalog.nl/) and the Dutch Commit-to-Data initiative (http://www.dutchdigitaldelta.nl/big-data/over-commit2data) (grant no: 628.009.001).

Printed and bound by Ipskamp printing.

ISBN: 978-94-6473-443-0

An Agent-based Approach to the Governance of Complex Cyber-Infrastructures

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. ir. P.P.C.C. Verbeek
ten overstaan van een door het College voor Promoties ingestelde commissie,
in het openbaar te verdedigen in de Agnietenkapel
op donderdag 4 april 2024, te 10.00 uur

door Mostafa Mohajeriparizi
geboren te Kerman

Promotiecommissie

Promotores:	prof. dr. T.M. van Engers	Universiteit van Amsterdam
	prof. dr. S. Klous	Universiteit van Amsterdam
Copromotores:	dr. G. Sileno	Universiteit van Amsterdam
Overige leden:	prof. dr. ir. C.T.A.M. de Laat	Universiteit van Amsterdam
	dr. P. Grosso	Universiteit van Amsterdam
	dr. L.T. van Binsbergen	Universiteit van Amsterdam
	prof. dr. A. Omicini	University of Bologna
	prof. dr. M.M. Dastani	Universiteit Utrecht
	prof. dr. K.V. Hindriks	Vrije Universiteit Amsterdam
	prof. dr. ing. L.H.M. Gommans	Universiteit van Amsterdam

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

to everyone

who has ever taught me something

Contents

Acknowledgments **xi**

1 Introduction **1**
1.1 Motivation and Research Questions 6
1.2 Approach and Scope of Dissertation 8
1.3 Research Context and Collaborations 12
1.4 Structure of the Dissertation 13

2 Developing a Scalable MAS Framework **15**
2.1 Introduction 15
2.2 Background 16
2.2.1 Agent Oriented Programming 16
2.2.2 Belief-Desire-Intention (BDI) Model 17
2.2.3 Actor Model 17
2.2.4 Related Work 18
2.3 AgentScript Cross-Compiler (ASC2) 19
2.3.1 ASC2 DSL 19
2.3.2 ASC2 Run-time Architecture 20
2.3.3 Translation Method 24
2.3.4 Tools for Execution 31
2.4 Performance Analysis 31
2.4.1 Token Ring 32
2.4.2 Chameneos Redux 35
2.4.3 Service Point 38
2.5 Discussion 40
2.5.1 Performance 40
2.5.2 Language 41
2.5.3 Execution and Parallelism 41
2.5.4 Access to the Lower-Level Language 41

2.5.5 Communication 42
2.6 Conclusion 42

3 Transparent Decisions in Social Actors: Preferences 43
3.1 Introduction 43
3.2 Background 44
3.2.1 Abstract Plans in BDI Agents 44
3.2.2 Preference Languages 45
3.3 Method 47
3.3.1 AgentSpeak(L) Agents 47
3.3.2 Abstract Events, Abstract Goals 49
3.3.3 CP-Nets and CP-Theories 51
3.3.4 Goal Refinement via Preferences 55
3.4 Implementation 58
3.5 Discussion and Conclusion 61

4 Interoperability and Automated Tests: DevOps 63
4.1 Introduction 63
4.2 Verification of (Multi-)Agent Systems 65
4.3 Approach 67
4.3.1 Testing Approach 68
4.4 Illustrative Example 70
4.4.1 Unit/Agent Testing 70
4.4.2 Coverage 73
4.4.3 Integration/System Testing 74
4.4.4 Continuous Integration 75
4.5 Discussion and Conclusion 76

5 Introducing Norms to Agents: Normative Advisors 79
5.1 Introduction 79
5.2 Core components 81
5.2.1 Intentional agents 82
5.2.2 Norms and Normative (Multi-Agent) Systems 82
5.3 Normative MAS via Normative Advisors 85
5.4 Implementation 87
5.4.1 Normative advisor architecture and decision-making cycle . 88
5.4.2 eFLINT norm base implementation 89
5.4.3 Spawning and interacting with advisors 91
5.5 Discussion 93
5.6 Conclusion 96

6 **Example 1: Coordination in MAS via Norms** **97**
6.1 Introduction . . . 97
6.2 Background . . . 98
6.2.1 Compliance Management Framework . . . 99
6.3 The Model of DMPs . . . 100
6.4 Executable Model of Data Market-Place . . . 102
6.4.1 Implementation of the Model . . . 103
6.5 Model Execution and Discussion . . . 107
6.6 Conclusion . . . 111

7 **Example 2: Qualitative, Quantitative, and Normative Reasoning** **113**
7.1 Introduction . . . 113
7.2 International Humanitarian Law rules . . . 114
7.3 The Model . . . 114
7.4 Decision-making . . . 116
7.5 Example Model of Normative Autonomous Devices . . . 117
7.5.1 Scenario . . . 117
7.5.2 Implementation . . . 118
7.6 Model Execution and Discussion . . . 123
7.6.1 Execution Results . . . 125
7.6.2 Discussion . . . 126

8 **Conclusions and Future Work** **129**
8.1 Motivation . . . 129
8.2 Summary . . . 130
8.3 Research Results . . . 132
8.4 Discussion, Limitations and Trade-offs . . . 138
8.5 Future Works . . . 141
8.5.1 Macro System Modelling . . . 141
8.5.2 Alternative Agent Reasoning Approaches . . . 142

Bibliography **145**

Samenvatting **163**

Abstract **165**

Acknowledgments

The work on this thesis started more than four years ago. Coming from a technical study and work background it was both a daunting challenge and an amazing journey to be a part of the academic community, following research questions I found important and interesting that sometimes seemed like endless roads.

Therefore, I would like to firstly thank Prof. Tom van Engers for accepting me as a PhD candidate and supervising me through my research, and having an indelible effect on my way of thinking that far exceeds this dissertation. I would also like to thank Giovanni Sileno, for the numerous compelling interactions and conversations many of which echo through every chapter of this dissertation.

I also should thank members of our research group(s) who were endlessly kind and welcoming towards me from the day I started. A special thanks to Prof. Cees de Laat that in my mind is the definition of a motivating and supportive leader, Thomas van Binsbergen that always had an interesting future research idea to share, and Ana Oprescu that seemed to have an answer to every problem.

Finally, words can not express my gratitude towards my ever-supportive partner and companion Shabnam, without whom this thesis, and most other good things in my life could never be possible. To my little Rasa, as you begin your own educational adventure, remember that your love and presence have always been my greatest motivation and source of strength. I eagerly look forward to witnessing your journey and achievements, wherever they may be.

Amsterdam
September 2023

Mostafa Mohajeri Parizi

Chapter 1

Introduction

Systems running on cyber-infrastructures are getting more integrated in all activities of our daily life and have increasingly more impact on society. Consequently, there is a growing need for approaches making sure that these systems are operating in a way that is more in alignment with societal norms (eg. as expressed by policies, rules, regulations). From an engineering point of view, this means that norms and policies should be considered as part of infrastructure development and maintenance cycles by designers. On the other hand, because infrastructures affect society, by extension, they also affect how society is governed, so they should be taken into account when policymakers and governance bodies are analyzing existing norms or implementing new policies and services.

Taking modelling and model-execution as principal instruments to analysing a system, intuitively, requires experts to be able to create executable models of all these elements: of the infrastructure, of relevant norms, and of the social setting, including its actors. The expressivity and flexibility required to encompass these aspects are not trivial. It is no longer just a model of an infrastructure, of a social setting, and of a set of norms, but, it is an infrastructure that is utilized by social actors and governed by norms; a social setting that utilizes and monitors the infrastructure, and decides upon, modifies, analyzes and enforces the norms; and norms that regulate the infrastructure and the social setting, and have impacts on both.

The purpose of this thesis is to closely study the interactions between these concepts, and search for fundamental gaps in current methodologies utilized by system designers, policymakers, and governance bodies. The thesis elaborates on approaches for creating more expressive and tangible models, also flexible enough for specifying different scenarios and use-cases in different domains. To propose realistic methods, aside from the theoretical analysis, proof-of-concept tools have also been developed or adopted at each step and will be presented with the thesis.

Social Norms and Behavioral Patterns The definition of norms for specific use cases and domains is specified in corresponding chapters, but, it is valuable to set the tone on the complexities of this concept. The social life of a human is filled with objectively complex interactions, but somehow we as humans can navigate these dynamic and nondeterministic interactions with a sufficiently low allocation of processing resources. A concept that aids us in doing so is norms, by defining what should and should not be the case and what is expected to happen or not to happen in certain contexts and under certain conditions, norms drive, or a least suggest, which actions to engage with in a complex world.

The fascinating thing about norms is that in any given situation, all parties involved are (somewhat) internally aware of the processes involved in that situation and they can act accordingly. We can even deduce these (most probably) correct processes in face of highly dynamic and nondeterministic situations with a multitude of variables. It seems like somehow we have converged very closely to a set of abstract, parameterized, context-based, and flexible scenario templates for interactions defined by norms that we use regularly as a coordination mechanism in society.

These scenario templates have placeholders for different roles each with its own script, and most of the time we can (almost) precisely interpret a situation, select a relevant and applicable template and instantly fill the parameters and role placeholders with qualified values and infer some sort of decision tree that tells us how interactions can and should move forward. We even monitor and observe events and (own or others') actions and qualify them as traversal steps in these scenarios. Furthermore, concerning our part in a scenario, we seem to have internalized behavioral specifications or partial plans that allow us to enact and embody the role that we have filled with ourselves.

Interpreting the context, selecting a relevant and applicable scenario template, filling in the variables, and qualifying observation are the main components of utilizing scenario templates defined by norms.

Let us take a simplistic example that is later on used in this thesis: a sale transaction. Regardless of the specific contexts, the norms for a typical sale transaction specify (almost globally accepted) predefined roles (buyer/seller), variables (item/price) and actions (offer/accept/pay/deliver) related to it.

For an example of interpreting the context and selecting an applicable scenario, take for instance in-person shopping versus online shopping; while there are obvious differences in the variables and actions involved, we can think of them as the same thing: a sale transaction, with the same abstract template. For an example of filling in the variables, we can easily qualify actors and put them in proper roles: the role of the seller in a sale transaction can be filled with ourselves, another person, a company, or more pertinently for a cyber-infrastructure context, even an automated software in the form of a webshop.

The same level of abstraction also applies to qualifying observations and expectations: receiving cash from an in-person buyer can be qualified as the

same action as a notification of a transfer from a banking infrastructure, and the expectation of receiving an item in the next few moments from an in-person seller is the same as the expectations of receiving an oversea delivery in the next few weeks.

In general, scenario templates defined by norms are highly dynamic and flexible. During an interaction, we are constantly monitoring the scenario for deviations from what we expect to occur. Some deviations can be negligible: while somewhat unexpected, they do not change the context of the interaction completely, like not receiving a delivery precisely when it was supposed to arrive. However, some deviations completely change the context, like never receiving an item you paid for and not being able to contact the seller.

What is interesting about deviations is that we even have modular and pre-defined contingency plans for when a concrete scenario is not played according to the expectations of its template. These plans can vary from attempting to fix the concrete scenario, like waiting a bit longer than expected for delivery, to revising some beliefs that resulted in the unexpected situation like removing a particular actor from the qualifications that make them suitable for the seller role to even rewriting the qualification or behavioral rules altogether like never buying anything from any unrecognized webshops.

Regulations, Compliance, and Governance Regulations are a form of norms that are (partially) concretized, written as a normative text, and enforced by some authoritative body. Cyber-infrastructural systems that affect members of a society ideally should be fully compliant with the norms and regulations of that society. There are multiple ways to ensure compliance in a system, depending on its size and complexity, the norms that govern it, and the governing bodies that are concerned with monitoring and regulating it. Some policies can be operationalized in terms of access control.

For example, some infrastructures with pre-defined and static internal policies can be developed by hard-coding these policies in their operations. Intuitively, this is the most efficient approach, but also the least scalable and maintainable. Some systems use run-time policy-making protocols like LDAP to make sure the infrastructure is running in a compliant manner by e.g., taking permission from policy enforcement points that take advice from policy decision points which in turn retrieve policies from policy information points. Then, by modifying the policies in the policy information points, the administrators can control the enforced policies without changing the infrastructure itself.

These approaches can be referred to as *compliant by design*, and are arguably suitable for many situations. That is, until the point that norms in question become more complicated than simple (eg. non-conflicting, non-contradicting, easy-to-interpret) regulative norms defined in terms of deontic notions of permissions, obligations, and prohibitions, implemented in the form of *ex-ante* access rules. As

is observed in the literature –and illustrated with multiple examples in this thesis – norms become more complex much sooner than anticipated by most studies, and in fact, are vastly simplified in current compliance-checking approaches.

Norms are more than a set of formal rules extracted from a legislative text: they emerge from multiple sources with different strength, and they require interpretation to be encoded, and qualification to be applied within a social context. Furthermore, they continuously adapt, in both expression and application [18]. Additionally, in any given context multiple normative sources may be concurrently relevant, and/or multiple interpretations of the same normative sources may be available (e.g. retrieved from previous cases), and these may be possibly conflicting. Finally, not all regulations are norms about controlling actions *ex-ante* , i.e., prior to the performance of the action; instead, they may be rules about what ought to be *ex-post*, ie. if certain actions have been performed already.

Furthermore, norms are traditionally distinguished between *regulative* and *constitutive* norms [144, 21, 149]. Regulative norms regulate behaviors that exist independent of the norms and are generally prescribed in terms of *permissions*, *obligations* and, *prohibitions* (e.g. traffic regulations). Constitutive norms determine that some entity (e.g. an object, a situation, a certain agent, a certain behaviour) “counts as” something else, creating a new institutional entity that does not exist independently of these norms. (For example, the concept of marriage, or money as a legal means of payment). Constitutive norms are particularly relevant for qualification acts. The concept of institutional power is also particularly relevant in the context of constitutive norms, as it is used to ascribe institutional meaning to performances (e.g. raising a hand counts as a bid during an auction).

The intrinsic complexities of dealing with norms, alongside the fact that cyber-infrastructural systems also include human actors and institutions that may be not fully compliant (and arguably, not even benevolent), entails that such systems can not be fully compliant by design. Instead, such systems require *governance*, which includes creation of new policies and regulations or modifying existing ones, promoting policy goals, providing supportive services, monitoring, analysing and imposing punishments and sanctions where needed. Intuitively, if components of the system can be feasibly compliant by design, that is a desirable attribute. However, when the governed system has rapidly changing technological components in it, there will be a need for approaches in adaptation of the governance that can keep up with these changes, and this thesis argues that modelling is a feasible approach.

Models of Agency Software systems are generally considered to be void of real agency (specifically in legal terms) and are controlled through specific control structures. Even if those structures consist of probabilistic flows that can change as the system interacts with its environment (like in the case of policies issued by reinforcement learning) they still have a control structure. This

assumption becomes fuzzy when it comes to computational models of social actors. These models need in principle to take agency into account: they are executable models that should represent humans or organizations, yet, without access to fully developed artificial general intelligence, it is hard to claim that a software agent has agency.

Then, the question is: how do we model such actors? There are two main approaches used in the field to model agency: (1) modelling agents in terms of variables and mathematical functions, and, (2) modelling agents with means of agent-oriented programs, often in the form of logic-based, rational behavioral specifications, that often are deployed in a Multi-Agent System (MAS).

The first approach has many advantages. Firstly, mathematical models are often more efficient in execution, resulting in them being more scalable. Also, defining agents in terms of a few variables makes them easier for experimentation, particularly in simulations. Finally, it also makes them much easier to manipulate at run-time, to simulate adaptation and learning processes.

In general, when the purpose of a research is to study the outcome of social phenomena in certain settings without regards to the model of the input agent, mathematical models are suitable. Example of application of these approaches is a simulation of disaster response and relief, where the purpose of the study is to find the optimal settings or infrastructures to minimize damage. In these cases, the agents —social or infrastructural— are generally considered to have mathematical specifications that encompass a statistically realistic representation of how a civilian, a disaster response team, or an infrastructure will or should act in case of an incident or natural disaster as their utility function [4, 107, 37], keeping higher level concepts like purpose or intentions implicit in the models.

Instead, where mathematical models fall short and programmable models are advantageous is in studies where expressivity and transparency of the models are paramount. In these cases, even if the execution is less efficient and adaptability is harder to achieve, it is still worth to have self-explanatory and readable models of agents that have explicit notions of agency (such as intentions), and act in a configurable (potentially highly heterogeneous) social setting. These are the cases where the purpose of the study is to analyze the agents and the society of agents without much regard to optimizing an outcome, and often there is no obvious and well-defined utility function to optimize anyway.

This research intrinsically falls into the second category category, and, as it will be presented in later chapters, it will build upon the Belief-Desire-Intention (BDI) model of agency [140] to model agents. By specifying agents in terms of human-related attitudes, BDI agents are suitable in modelling social agents (computational or not). Interestingly, the majority of the concepts defined in the previous section about norms, like the presence of behavioral specifications that can be matched and concretized in certain situations, or having contingency plans for failures are all already (partially) identified by BDI models, to various degrees of success.

1.1 Motivation and Research Questions

The overarching motivation of this research lies in the creation of a digital data market-places (DMP), with a special concern to the field of logistics data. The need for data is exponentially increasing in all research and industries, and an environment like a DMP can provide different parties a place to share (or buy and sell) scientific or corporate data with each other. Although the presence of such environment can vastly improve the efficiency of data-oriented research and even prevent issues like data monopolies, there are certain risks involved, such as privacy issues, security issues, competitive corporate advantages, legal and ethical challenges, and governance and accountability concerns.

Furthermore, as more governments (and other authoritative bodies) are implementing regulations to govern data transactions, such sharing environment needs to be compliant with these regulations. Additionally, actors in these markets (organizations, companies, individuals) may have ad-hoc contractual agreements about data-sharing, and they may also have internal policies (e.g. user agreements about how they can share data). The complexities of the market-places alongside the requirement for compliance results in a need for appropriate governance instruments, only at cyber-infrastructure level.

A crucial part of governance are policies. They provide a framework for decision-making, establishing guidelines and rules, and guiding the actions of individuals and organizations. Policies are generally about setting directions and goals for a system, regulating behavior, managing resources and risks, and, promote accountability.

In the domain of data-sharing, policies have more specific roles. From the perspective of privacy, policies may specify how personal and sensitive data should be handled, defining the conditions for data anonymization and de-identification. Policies can also ensure compliance with relevant laws and regulations related to data protection, intellectual property, consent, and confidentiality. They may define appropriate criteria for data access, such as applicable intentions for a certain use of data, and performance of appropriate actions, like obtaining consent.

In principle, policies can promote fair and equitable data sharing, and establish rules for data usage, reuse, and retention [130]. They address ethical considerations such as fairness, transparency, and accountability in data-sharing practices. Policies may also be about data security, they establish measures to protect data from unauthorized access, breaches, and cyber threats. They outline security protocols, encryption standards, access controls, sufficient monitoring and data breach notification procedures.

Another important aspect of policies and policy-making involves engaging relevant stakeholders, including individuals, industry representatives, researchers, and policymakers, to gather input, address concerns, and ensure that diverse perspectives are considered in shaping the data-sharing policies.

The importance of policies in data-sharing intuitively results in the crucial role

of policy-making. In data-sharing, policy-making generally refers to the process of developing and implementing rules, guidelines, and frameworks that govern the collection, storage, access, use, and sharing of data and many other aspects of data-sharing. It involves the creation of policies, regulations, and standards that define the rights, duties and responsibilities, permissions, obligations, and power relations of various stakeholders involved in data-sharing, such as individuals, organizations, and governments. This leads to the overarching goal behind this dissertation and other research efforts: *Defining approaches, methodologies and tools for policy-making in the data-sharing domain*.

In general, policy-makers are required to take into account the effect and impact of their policies. For long-term and complex policy matters, it has been argued that this is only feasible with mathematical or computer-based models [28]. Modelling can play a crucial role in policy-making by providing insights, predictions, and evaluations that inform the development, implementation, and assessment of policies. Modelling helps with understanding complex systems allowing policy-makers to gain a better understanding of systems and their dynamics. Modelling enables scenario analysis by simulating different scenarios and predict the potential outcomes of policy choices resulting in more optimized policies with more balanced between goals. Finally, models can facilitate more transparency and better policy communication with stakeholders and even system designers.

There are different types of modelling used in the different communities to assist in policy-making. These approaches vastly differ in their focus, methodology, and, modelling time-frame. From macroeconomics to system dynamics to agent-based modelling, each approach has its own use cases in specific domains, and there are arguments to use one or a combination of these approaches in different use cases [143, 136, 71, 67]. Agent-based models are suitable for policy-making as they define the behavior of individuals to build emergent trajectories of the system as a whole. In a real system, where policies are based on top-down assumptions of behavior, many changes occur bottom-up from individual actors' actions [67]. This dissertation focuses on utilizing agent-based models for policy-making; then, the more concrete goal of this research would be: *Defining approaches, methodologies and tools for policy-making in the data-sharing domain based on Agent-Based Modelling*.

While this dissertation focuses on data-sharing, it is not only data-sharing use-cases that are in need of such methods for policy-making and system design. More aspects of our life are being *controlled* by automated processes; visa applications, job applications, credit placements, mortgage and insurance are just a few examples and one can only imagine that the list will continue to grow as time goes on. Even if data-sharing regulations are a relatively new phenomena, when taken in a broader sense, there are already regulations implemented for other domains that are now becoming more automated, and when the decisions are made in an automated manner, then the goal will expand to defining policies for any cyber-infrastructural system with respect to arbitrary (regulative and constitutive)

norms. This dissertation takes the aforementioned goals, by assuming (agent-based) modelling and model execution as a principal step in system design and policy-making:

> **Main Research Question:** *How can we model a norm-governed cyber-infrastructure for the purpose of policy-making?*

1.2 Approach and Scope of Dissertation

The first step in defining the scope of this work is to break the main research question structurally by further defining what needs to be modelled: a norm-governed cyber-infrastructural system. The main components are norms, social setting, and software/infrastructure that create the system as a whole. We also need to define what are the requirements of these models to be suitable for policy-making. Then, the main research sub-questions should intuitively become how to model each of these three components. But, unfortunately it is not that simple as these concepts are not at the same level of abstraction for the purposes of this research.

Take the concept of norms, while modelling them is essential for the whole picture, it is still social agents that act as the governance bodies that regulate other entities of the system, and in effect, it is social agents that are being governed, i.e., it is very hard to talk about norm-related concepts (e.g. sanction) without referring to human-related concepts (e.g. intention). Indeed, it is rare to see an infrastructure being punished for non-compliant actions without referring to a social agent (a person, company, institution, organization) as the main liable entity for those actions, or, it would be untenable to say a piece of software or infrastructure is imposing a sanction without the presence of a social agent (an organization, consortium, government) holding the power to impose that punishment. In effect, norms as they are studied in this research —as it is often the case in real life— do not have isolated meaning by themselves and without the social setting. The same logic goes for software and infrastructural components, while being another important part of the system model, it is still their usage by, and their effects on social agents that is being scrutinized.

In summary, in the context of this dissertation, social agents are the glue that hold the models together and gain therefore a primary role. The following requirements can be identified with respect to policy-making:

1. Effective social simulation that is suitable for policy makers requires modelling the individual decision-making process given subjective social norms, individual preferences, and policies; in other words, we require highly expressive agents in terms of cognitive capabilities.

2. The agent models should have high scalability; this is important in this context to grant the model designers the freedom to create models with a high number of agents and interactions.

3. The framework should be modular. Modularity is crucial because there are many different theories in the literature about norms, preferences, personal policies, and other aspects of the agents. A highly modular agent architecture allows us to experiment with these theories without the need to hard-code them into the reasoning cycle of the agent.

This gets us to the first research question:

> **Research Question 1:** *How can we create expressive, scalable and modular models of social agents?*

The Belief-Desire-Intention (BDI) model of agency has been identified in the literature as a suitable approach to create software agents with the ability to reason about norms [68, 60], to have and apply preferences [162], and that can be utilized effectively in policy-making [67]. While the reasons behind this choice will be thoroughly explored through this work, the approach adopted to agent-based programming in this thesis will be to rely on the BDI model to create the social agents. Although there are multiple BDI frameworks introduced in the literature, after much deliberation and testing, presented in Chapter 2, it turned out they did not meet the requirements of scalabilty, modularity, and, interoperability.

Alongside with this thesis, a BDI (and MAS) framework called AgentScript Cross-Compiler (ASC2) was designed and developed, mainly with the scalability and modularity requirements in mind. ASC2 relies on a programming language based on the AgentSpeak(L) but does not assume a hard-coded reasoning and decision-making cycle for agents; in fact, almost every aspect of the agents is programmable.

Firstly, ASC2 utilizes actor-oriented programming via the Akka actor framework: each agent consists of multiple actors, each with their own role, able to communicate through internal messaging, effectively making an agent a modular actor micro-system in itself. Secondly, the design of ASC2 follows software engineering best practices and methodologies like Dependency Injection and Inversion of Control. By using Dependency Injection, most components and sub-components of an agent can be sent to it [NOT CLEAR!] as potentially customized dependencies. Furthermore, with Inversion of Control, it is the lower level components that are mainly controlling the nuanced [NOT CLEAR] execution cycle when higher level components only define abstract control flows. These design decisions also result in scalability of the framework: the ASC2 framework is introduced, analysed, and, benchmarked from an engineering perspective in Chapter 2.

Next, there is the issue of how to model software and infrastructural entities. Software and infrastructure components are one of the main components of our

models, and being able to include them is essential. However, the question of how to model the software components falls out of the scope of this thesis as there is an extensive body of research on software modelling. Instead, the focus here is on integrating software components with the software agents. Steps are taken to provide a high level of interoperability for social agents modelled in ASC2 to virtually any type of model of software and infrastructural entities that are being used by the designer and policy-maker, including the actual real entities.

> **Research Question 2:** *How can social agents utilize software and infrastructural models or entities?*

One of the bigger drawback of existing BDI frameworks that was identified during this research, which was also a motivation in creating ASC2, is the lack of straightforward ways the have agents interacting with arbitrary environments. Here, environment means an arbitrary piece of software external to the agent. This can be a communication interface, an execution environment, or any other type of software component. In the data-sharing context, for example, this environment is the data-sharing infrastructure — or a model of it– that the agents need to communicate with through some arbitrary API. Interestingly, this is hardly recognized in the classic MAS literature as a challenge or requirement, and therefore, the lack of it as a drawback. However, by reviewing more recent literature we can observe that there are multiple works [138, 46, 121] that try to interface existing BDI frameworks with modern software components and architectures like web services or micro-services.

ASC2's design however gives the agents the advantage of simple interoperability with the external environment. As it is addressed in Chapters 4 and 5, after compilation ASC2 agents are technically JVM programs, meaning they are indeed interoperable with any system that (for example) a Java program can utilize or communicate with. This results in much more effective modelling cycles, as there is far less concern about connecting the models to a real system or interfaces.

Next is the notion of norms, modelling them including social norms, regulations, contractual agreements, internal policies is an important part of the modelling that this thesis aims at. However, while different approaches and ideas about modelling norms are explored in this work, defining any novel approach to do so falls out of the scope of this thesis for both theoretical and practical reasons. The practical reason is mainly that other threads of dedicated research were being performed on this matter within the research group that both affected and utilized this work. The theoretical reason is that there are various levels of abstraction that norms can be modelled for different use-cases and this work stays agnostic to which approach is the most suitable. However, the question that is studied is the concept of norms from the perspective of social agents:

> **Research Question 3:** *How can social agents reason with, and, about norms?*

There are multiple works in the literature about the interaction of agents with norms [68, 33, 156, 49]. The main concern of this dissertation on this question is to identify the interactions between social agents and norms, including the interpretation and qualification of events between institutional and physical realities, understanding compliant and non-compliant behaviors and the relation between compliance and autonomy. This is addressed in chapter 5 with introducing the concept of *normative advisors*, a flexible approach that can be utilized by social agents to have an understanding of normative positions in their environment without reducing their autonomy.

Modelling agents for the purpose of policy-making requires to reproduce to a certain extent decision-making constructs similar to those observed in human institutions. Furthermore, for *traceability* and *explainability* reasons, decision-making that precedes actions is as relevant as the behavior. Although BDI models — and more specific to this work, AgentSpeak(L)-like models— are designed with traceability and explainability as a first-class requirements [139], when it comes to more complex forms of decision-making – like the issues that manifest in conflict resolution– the question of modelling objectives, desires, and preferences of agents are not trivial issues [68, 162, 123].

In the process of this research, it was observed that by adding norms and normative reasoning into the agents, these concepts become even more important. Norms in every form can be conflicting, and such conflicts can rarely be resolved by typical approaches [19, 176]. Furthermore, norms can also be conflicting with the goals that an agent is trying to achieve [34], how should an agent program behave in face of conflicts? On what basis should it make decisions in such situations? There is no clear-cut answer to these questions, as it will depend on the context. This dissertation addresses this by taking the stance that it depends on the preferences and desires of the agent. It is the agent that should decide based on the context how to resolve a conflict, and the designer should be able to program such notions. The issue addressed here is not about what is the *most appropriate* behavior of the agent as the consequence of its decision-making and conflict resolution; that will depend on what the designer is modelling. Instead, it is how such decision-making can be expressed in a traceable manner:

> **Research Question 4:** *How can we make the agents' decision-making traceable and explainable?*

This question is addressed in Chapter 3 by adding explicit preferences in the form of CP-Nets into the BDI model and the AgentScript programming language. With this approach, the designer can separate the concern between the procedural or the *how-to* knowledge of the agent program from the preferential or *what-to* knowledge. This makes the decision-making of the agents more transparent, which is a requirement in models that are created for the purpose of policy analysis.

The last issue addressed in this thesis is the practicality of utilizing agent-based models as part of real-world system design and policy-making, not only as they are

used in this work but the whole agent-based modelling community. It is always the case that accessibility and usability of the tools in a certain methodology is an important part of their adoption, it is hard and mostly not feasible to convince domain experts like programmers to utilize an approach if they need to also learn a whole new set of tools and ecosystems. This is also the case for utilizing agent-based models and has been a major concern in this work

> **Research Question 5:** *How to make agent-based modelling a usable approach for mainstream designers?*

In recent years, the tools created for design and development in mainstream software community are becoming more advanced and efficient. A few examples of such tools are IDEs, testing libraries, build tools, code coverage tools, code repositories, and DevOps system like Continuous Integration and Deployment (CI/CD) tools. Intuitively, it is advantageous to allow for utilization of these tools in agent-based modelling and model execution. As a side effect of its design, ASC2 programs can directly utilize any system or library available to any other software programs without the need for any additional piece of software, because, after compilation, ASC2 programs become standard JVM-based programs. This includes all the development tools listed above, and many others. In Chapter 4 it is shown how utilizing these tools, not only benefits agent and multi-agent system (MAS) communities in e.g., testing their designs with mainstream automated testing tools, but more importantly, allows agent models to be used as part of any software development process, be it for testing or any other purpose.

1.3 Research Context and Collaborations

Before starting the next chapter, I will try to put this dissertation into the context and environment that the research was conducted, highlighting the parallel research that had a connection to it and some of the events that affected it. The research was done in the Complex Cyber-Infrastructures (CCI)[1] research group and was funded by the project Data Logistics for Logistics Data (DL4LD)[2]. There were two other closely related projects: Secure Scalable Policy-enforced Distributed Data Processing (SSPDDP)[3] and Enabling Personalized Interventions (EPI)[4] that shared research with DL4LD. The following highlights some of the parallel research threads that were conducted, mostly by other PhD candidates and post-doctoral researchers, alongside this work. In the context of this dissertation, Agent-Oriented Programming approaches were utilized for developing Agent-Based Models. The counterpart to this are the mathematical models of agents which

[1] https://cci-research.nl/

[2] https://dl4ld.nl/

[3] https://cr-marcstevens.gitlab.io/sspddp/

[4] https://enablingpersonalizedinterventions.nl/

were simultaneously studied in the context of the group by Fratrič et. al. [82, 81]. Where this thesis tries to find approaches in enhancing models for developing policies, there were parallel works that focused on what these policies should be, including risk management and enforcement schemes by Zhou et. al. [174, 173]. Another thread of research, as it was mentioned in the introduction was about norm reasoning. The results of these works presented in [158, 157] are used throughout following chapters. Finally, there was a thread of work mainly on architectural design of Data Market-Places e.g., [145, 171] that share much of their requirements with this dissertation.

Finally, the majority of the time dedicated to this research overlapped with COVID-19 restrictions, and apart from "normal" issues, it meant there was little to no real interaction with the industry partners that were the initial stakeholders of this project. To provide full transparency for the reader the phrase "it turned out that it is not only data-sharing use-cases that are in need of such methods for policy-making and system design" in the motivation section, while still true, also partially means that it is considerably more challenging to study policies and policy-making in data-sharing without the presence of stakeholders that have access to data and are interested in sharing them. As a result, while this research was at a relatively high level of abstraction at its inception, for better or worse, became even more theoretical that it was intended to be —or I intended it to be—, but fortunately, my supervisors were already interested in the more theoretical side of the issue [159, 148] which greatly guided this research in every aspect. However, by interacting with the respective communities e.g., AI and Law, Normative Systems, and, Multi-Agent Systems, my observation was that there is an interest for more practical and usable approaches to bring the long-standing results of these communities closer to mainstream domains. For this reason, every chapter of this dissertation includes development of tools or proof of concepts that utilize fairly mainstream software ecosystems.

1.4 Structure of the Dissertation

The dissertation is organized as follows: Chapter 2 Introduces ASC2, an agent programming framework based on BDI that acts as both the technical and theoretical backbone of this research. Chapter 3 proposes an approach for embedding preferences in form of CP-Nets into ASC2 agents. Chapter 4 discusses methods for making a BDI modelling framework interoperable with mainstream software tools with an emphasis on testing. Chapter 5 focuses on adding norms into agent-based and multi-agent models, introducing the concept of normative advisors. Chapter 6 is an illustrative example of using the modelling framework, focusing on coordination in MAS via norms. Chapter 7 is yet another illustrative example which discusses mixing qualitative, quantitative, and, normative reasoning in the models. Finally, Chapter 8 offers the summary, conclusion, limitations, and, future works.

Chapter 2
Developing a Scalable MAS Framework

Agent-based modelling is a valuable tool in designing complex socio-technical systems. This chapter introduces an Agent-Oriented Programming (AOP) framework based on the Belief-Desire-Intention (BDI) model of agency called AgentScript Cross-Compiler (ASC2). There are multiple BDI frameworks already introduced in the literature. Prior to the development of ASC2, we tried to adapt and utilize several other frameworks. While there were different advantages and disadvantages to each of them, in the end, the main reason that resulted in designing and developing yet another framework was scalability, mainly in terms of development process which includes interoperability and maintainability of agent programs. None of the other frameworks allowed for seamless and effective interoperability between agents and other mainstream software and development tools. Also, most BDI frameworks have most of the agent's reasoning hard-coded into them, which limits experimentation on alternate theories.[1]

2.1 Introduction

Agent-based models have an intuitive mapping to behavioral descriptions, and for this reason, are extensively used for modeling and simulations of social systems. However, *agent-based programming* is not only relevant for simulation. Complex Cyber-Infrastructures like those used for data-sharing as digital marketplaces exhibit the double status of computational and social systems; regulating these infrastructures requires reproducing to a certain extent constructs similar to those observed in human reasoning (e.g. For which purpose is the agent asking access to the resource? On which basis is the infrastructure granting access?). For *traceability* and *explainability* reasons, decisions concerning actions need to be processed by the infrastructure as much as relevant operational aspects. Agent-based programming, by looking at computational agents as intentional agents,

[1]The material presented in this chapter refines and extends elements presented in [126].

provides this level of abstraction available *by design*. However, this raises concerns about how we can efficiently map logic-oriented agent-based programs into an operational setting, a problem motivating the present research.

This chapter introduces ASC2, a logic-based AOP framework in which agents are modular micro-systems run by actors. By getting inspiration from the design of previous works, the novelty of this framework is (a) ASC2 is a cross-compiler and (b) it relies on the Actor model, instantiating each intentional agent as an autonomous micro-system run by actors. The hypothesis behind this choice is that defining the agents via actors results in a more fine-grained modular architecture that is more effectively modifiable, and that the execution of agent-oriented programs is enhanced (in scalability as well as in performance) by relying on robust implementations of Actor models such as *Akka*. The goal of this chapter is to (1) introduce ASC2 and its DSL as they will be a basis for the rest of this thesis (2) illustrate the novelty of the the ASC2's structural architecture and inner workings, and (3) provide a comprehensive qualitative and quantitative comparison between ASC2 and other similar frameworks.

To evaluate the feasibility of this approach for future developments, the first implementation of ASC2 based on Akka running on JVM is compared with three other relevant AOP frameworks (Jason [26], ASTRA [65] and Sarl [65]) by means of 3 benchmarks (token ring, chameneos redux and service point), known to capture relevant patterns in concurrent applications. This performance evaluation shows that despite its relative youth and the new implementation approach, ASC2 is competitive against existing frameworks, making it worthy of further investigation.

The chapter proceeds as follows: Section 2.2 provides some background on relevant concepts and related works. Section 2.3 presents the ASC2 framework. Section 2.4 reports on the empirical experiments comparing ASC2 with other frameworks. Section 2.5 compares the frameworks qualitatively. A note of future developments ends the chapter.

2.2 Background

2.2.1 Agent Oriented Programming

Agent-Oriented Programming (AOP) is a programming paradigm that uses mental attitudes to model autonomous computational agents. Introduced in 1993 by Shoham [147], it has attracted increasing attention ever since and is believed to provide an effective abstraction to approach complex software systems (e.g. [141]). In the beginning, it was presented as a specific version of Object Oriented Programming (OOP): whereas object classes contain arbitrary components, agent types share the same types of mental states and of structural relationships/mechanisms involving those states.

2.2.2 Belief-Desire-Intention (BDI) Model

BDI frameworks are usually described in terms of an *agent theory* and an *agent computational architecture* [78]. The agent theory usually refers to Bratman's theory of practical reasoning [31], describing the agent's cognitive state and reasoning process in terms of its *beliefs*, *desires* and *intentions*. Beliefs are the facts that the agents believe to be true in the environment. Desires capture the motivational dimension of the agent, typically in the more concrete form of *goals*, representing procedure/states that the agent wants to perform/achieve. Intentions are selected conducts (or *plans*) that the agent commits to (in order to advance its desires).

The agent architecture varies depending on the platform. Taking for instance the BDI platform Jason [26] as the biggest inspiration for ASC2, it consists of perception and actuation modules, a *belief base*, *intention stacks*, and an *event queue*. The BDI execution model in Jason, describing the agent's reasoning cycle, can be summarized as follows:

1. observe the external world to update the internal state (perception);
2. update the event queue with percepts and exogenous events;
3. select events from the event queue to commit to;
4. select plans from the plan library that are relevant to the selected event;
5. select an intended means amongst the applicable plans for instantiation;
6. push the intended means to an existing or a new intention stack;
7. select an intention stack and pull an intention, execute the next step of it;
8. if the step is about a primitive action, perform it, if about a sub-goal post it to the event queue.

As exemplified by this description, an essential feature of BDI architectures [140] is the ability to instantiate plans that can: (a) react to specific situations, and (b) be invoked based on their purpose. Consequently, the BDI execution model naturally relies on a *reactive* model of computation (cf. *event-based programming*, typically used for user interfaces). Multiple programming languages and frameworks have been introduced based on the BDI model, as AgentSpeak(L)/Jason [139, 26], 3APL/2APL [54], GOAL [99] and IMPACT [69].

2.2.3 Actor Model

The Actor model, introduced in [98], is a mathematical theory that treats *actors* as the primitives of computation [97]. Actors are essentially reactive concurrent entities. When an actor receives a message it can:

- send messages to other actors;
- spawn new actors;
- modify its reactive behavior for the next message it receives;

Originally proposed as a tool for the theoretical understanding of concurrency, the Actor model serves now as the basis of several production-level solutions for distributed and asynchronous systems, and for reactive programming. These solutions include: Akka [91], a library developed for the JVM environment, enriched by a strong community with multiple complementary tools for distributed environments and stream processing; the C++ Actor Framework (CAF) [40], a library for creating concurrent programs in C++; Pony [42, 43], an actor language for building robust parallel systems by providing data-race-free isolation for actors. A comprehensive overview and benchmark of these works can be found in [17].

2.2.4 Related Work

Multiple AOP and BDI frameworks have been introduced proposing diverse approaches towards language, execution model, reasoning process, etc. Jason [26] is plausibly the most known (e.g. it is the most used choice in the Multi-Agent Programming Contest [55]), and has been constantly developed in the last 15 years. It is implemented in Java and is essentially an *interpreter* for a logic-based DSL, namely an extended version of AgentSpeak(L) [139]. Two recent frameworks inspired by Jason are Pyson [3] and LightJason [6]. Pyson is an interpreter implemented in Python and uses MapReduce technology as execution infrastructure in order to achieve better scalability specifically w.r.t. the number of agents. LightJason is a BDI framework based upon a variation of AgentSpeak(L) and whose interpreter aims to improve the scalability of Jason by implementing a concurrent platform following best practices in software engineering.

ASTRA [65] is yet another framework inspired by AgentSpeak(L)/Jason and is also implemented in Java, but, unlike Jason, it is not an interpreter. ASTRA relies on a *compilation* approach: through a build pipeline the DSL is first translated to pure Java code and then the Java code is compiled to byte code for execution. In contrast, the Sarl [141] framework has not been introduced as a BDI platform, and then it does not use the same abstractions. Nonetheless, it is an AOP framework written in Java that also uses compilation, and for these reasons, it is relevant to the current study.

Although several AOP/BDI frameworks have been introduced in recent years (all hinting at problems of scalability), there is a small amount of empirical data available about how they perform in comparison to each other. The most notable exception is [36], in which multiple actor and agent frameworks (2APL [54], GOAL [99], Jason and Akka) are benchmarked. Their results showed that Jason outperformed other BDI frameworks by far and scaled almost on par with Akka. However, at that time (2013), none of these newer frameworks had been introduced yet, and Akka had not the support it has today. Strangely enough, none of these new AOP frameworks has the Actor model at their foundation. The present chapter aims to investigate part of this gap.

2.3 AgentScript Cross-Compiler (ASC2)

This section provides an overview of ASC2, however, as this framework is central to this thesis, detailed descriptions of different parts related to specific concepts are presented in the respective sections: Chapter 3 introduces the addition of preferences to the DSL, Chapter4 explores interoperability between ASC2 and mainstream software development tools. The ASC2 framework consists of:

1. a logic-based Agent-Oriented Programming DSL;
2. an abstract agent run-time architecture;
3. a translation method that generates executable models from models specified by the DSL;
4. tools that support the execution of models. We provide here a brief overview of these components.

2.3.1 ASC2 DSL

ASC2's DSL has a very close syntax to AgentSpeak(L) language and includes some of the extensions provided by Jason. The main components of the DSL are (1) initial beliefs, (2) initial goals, and (3) plan rules. Initial beliefs are a set of facts and inferential rules that are potentially non-grounded declarative rules, used to infer beliefs from beliefs. Initial goals designate the first intentions to which the agent commits. These can be used as a way to initialize an agent in their environment, and maybe start interacting with other agents.

Plan rules are potentially non-grounded reactive rules in the form $e : C \Rightarrow H$ that map different internal or external events e (e.g, *goal adoption*, *belief-update*) to a sequence of executable steps H called the *plan body* which the agent will perform in response to the event. Each plan also has a context condition C which is a boolean expression that represents when that plan is applicable. The high-level overview of ASC2's grammar definition is presented in Listing 1. A more detailed definition is presented alongside the translation method in Section 2.3.3

```
agent            → initial_beliefs initial_goals plans
initial_beliefs          → (term '.')∗
initial_goals    → ('!' atomic_formula '.')∗
plans            → (plan '.')∗
plan             → ( '@' atomic_formula )∗
                 trigger_event ( ':' context )
                 '=>' body_definition '.'
```

Listing 1: AgentScript's DSL grammar defenition

An example of an ASC2 DSL that shows part of the script for a domestic robot

is presented in Listing 2, where lines 2-4 are initial beliefs, line 6 is an inferential rule, line 9 is an initial goal and lines 12-15 define an example plan.

```
% initial beliefs and inferential rules
main(fish). soup(veg). wine(white).
restaurant(french).
at(home).

meal(S,M,W) :- soup(S), main(M), wine(W).

% initial goals
!go_order(french,meal(veg,meat,white)).

% plans
% P1
+!go_order(Loc,Meal) :
        restaurant(Loc) && not at(Loc) =>
        #move_to(Loc);
        !order(Meal).
...
```

Listing 2: An example script of ASC2 DSL

2.3.2 ASC2 Run-time Architecture

The run-time architecture of ASC2 agents can and should be inspected from both functional and structural perspectives. ASC2 is primarily motivated by AgentSpeak(L)/Jason as a starting point, and in its default setting it is designed to have (almost) the same functional architecture. However, from a structural perspective, ASC2 is novel in the sense that it utilizes the actor model to instantiate agents as actor micro-system. This Section, after a short description of the functional architecture and simple examples to get the reader acquainted with the DSL, focuses more on the structural architecture of ASC2. Then, in Chapter 3 where the functional architecture of ASC2 is extended, a formal and more complete description of reasoning in ASC2 is provided.

ASC2 Functional Architecture

An ASC2 agent consists of a set of beliefs B called belief base, a set of plans P called plan library, a set of events E, a set of intentions I, and three selection functions: S_E,S_O,S_I. When the agent receives an (internal or external) event or adopts a goal, it is added to E. The selection function S_E selects an event to process from E.

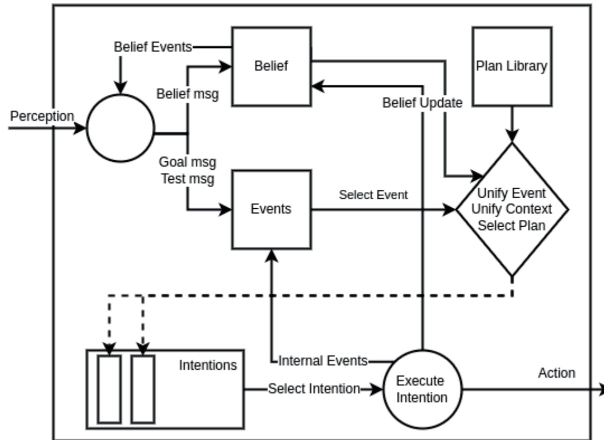

Figure 2.1: ASC2's functional architecture

Then, this event is matched with the triggering events in the heads of the plans in P. The plans that their triggering event unifies with this event are called *relevant plans* and their unifier is called the *relevant unifier*. Then for each relevant plan, the relevant unifier is applied to the context condition of that plan, and by querying against B a substitution is created such that the context is a logical consequence of B. The composition of relevant unifier with this substitution is called an *applicable unifier*.

As for each event, there could be multiple applicable unifiers, the selection function S_O chooses one of these plans or options, and applying the applicable unifier to that plan creates an instantiated plan, i.e, an intended means for the event which will be added to a new or existing intention. Then the S_I function selects an intention that will be executed. Figure 2.1 presents an overview of the functional architecture embedded in ASC2 agents.

Example 1 Take the example script in Listing 2, Suppose the agent receives and event in the form of:

```
!go_order(french,meal(veg,meat,white))
```

For this event, plan P1 is relevant with unifier:

```
{Loc/french, Meal/meal(veg,meat,white)}
```

Then applying this unifier to the context condition of that plan will result in the expression:

```
restaurant(french) && not at(french)
```

Which indeed is a logical consequence of the agents beliefs, meaning this plan is applicable with the same unifier as before (i.e., the query does not add any new substitutions). Then, by applying this substitution to the plan itself will result in:

```
+!go_order(french,meal(veg,meat,white)) :
    restaurant(french) && not at(french) =>
    #move_to(french);
    !order(meal(veg,meat,white)).
```

The steps in the body of the instantiated plan become an intended means for the agent to execute. The first step `#move_to(french)` is simply a function call. In ASC2, this could be any callable entity on the JVM's execution classpath. In this example, we are assuming that this call somehow relocates the agent to the location specified as the parameter. The second step is `!order(meal(veg, meat, white))` which is a sub-goal. In functional terms, a sub-goal is processed by the agent as an internal event that will follow the same plan selection process.

This process is better described in ASC2 as a reasoning flow instead of a reasoning cycle. This is because unlike what is typically the case in BDI frameworks, ASC2 does not implement an explicit synchronous reasoning cycle. As a result, the two selection functions S_E and S_I are asynchronous: when an event arrives, it will be processed in an asynchronous manner given there are free resources available to the agent. Or, when a new intent is created, it will be executed asynchronously given there are free resources available. In effect, the sets E and I are priority queues and S_E and S_I are sorting functions that define the priority of the entities in these queues. The default implementation for both is a simple first-in-first-out (FIFO) algorithm. A new event or an intention will be selected by their respective selection function when processing resources, namely threads from a thread pool become available.

ASC2 Structural Architecture

The structural architecture of ASC2 agents is based on the Actor model. Each agent consists of multiple actors with different roles: (i) an **Interface** actor, (ii) a **Belief Base** actor, (iii), an **Intention Pool** actor and (iv) $N \geq 1$ **Intention** actors. Each agent has also non-actor components: (1) a plan library, and (2) one or more belief bases. An overview of this architecture is presented in Figure 2.2.

The plan library of the agent consists of a set of plan rule objects in the form `{e,c,h}`, where `e` is an object that can be matched and unified with event messages to determine if a plan rule is relevant for that event, `c` is a term object that can be sent to the Belief Base actor to determine if the plan is applicable and `h` is a function representing the body of the plan.

The belief base(s) of the agent can be in practice any type of storage technology. To interface an arbitrary belief base into the agent architecture a translation

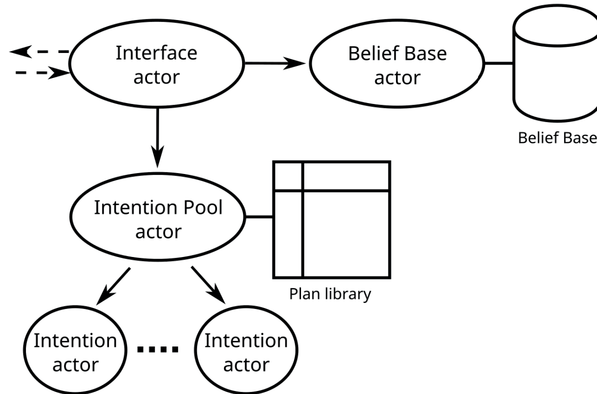

Figure 2.2: ASC2's structural architecture

function needs to be implemented for mapping the query messages into the queries of that belief base and vice versa, translating the responses into result messages.[2]

Interface Actor

The Interface actor acts as the main entity of the agent. It initializes the Belief Base and Intention Pool actors and then sends the initial beliefs and inferential rules to Belief Base actor as *assert* messages and initial goals to Intention pool actor as *achieve* messages. This actor is the only component of the agent that is accessible from the environment and the other agents: all incoming messages and events must go though this actor and any message sent from this agent will indicate the Interface actor as the sender of message. When the Interface actor receives a new message m, based on the type of the message it will either process it itself if m is a control messages, (e.g, *halt*), forward it to Belief Base actor if m is an assert message (e.g, *perception*) or forward it to Intention Pool actor if m is an achieve message (e.g, *request*).

Belief Base Actor

The Belief Base actor maintains the connection between other components of the agent and any data storage/reasoning engine that is used as the belief base. This actor accepts query messages (*retract*, *assert* and *unify*) and responds with result of the query. The technology of the data storage(s) is abstracted behind

[2]For the benchmarks presented in this work we used a lightweight open-source Prolog reasoning engine implemented in Scala called *Styla*, available at `https://github.com/fedesilva/styla`. The library was minimally modified and is available at `https://github.com/mostafamohajeri/styla`.

this actor and it can be changed by the programmer without affecting the rest of the framework. The specific implementation of the belief base is injected to the agent initialization time. Apart from processing queries, the Belief Base actor also feeds back *belief-update* events to the Interface actor. The semantics of when these events should be created are externalized to the core of architecture and can be programmable by the designer. This is the inversion of control as the belief base can control other components by sending messages to them.

Intention Pool actor

The Intention Pool actor receives events from the Interface actor and processes them. To process a received event `v`, the set of relevant plan rules `{e,c,h}` are selected from the plan library by matching and unifying `v` against `e`. Then these relevant plans are fetched from the plan library and sent to an idle Intention actor. The Intention pool actor can spawn N Intention actors, where the configurable number N dictates the number of concurrent intentions each agent can have at each instant. This actor uses a prioritized mailbox that sorts the messages based on the externalized programmable priority function S_E and a new event is processed only if there are idle Intention actors to forward it to. This mechanism makes sure that as long as there are no resources available, new events stay in the mailbox to be re-prioritized by S_E and when an idle Intention actor becomes available the event with the highest priority is processed [3].

Intention Actor

An Intention actor is a reusable unit of execution for the agent. It receives an event `v` alongside a set of plan rule objects `{e,c,h}` from the Intention Pool actor for execution. The execution consists of three phases: (i) the applicability of each plan rule is checked by sending a query message containing `c` to the Belief Base actor; (ii) from the set of applicable plans, one is selected by the externalized programmable function S_P for execution; (iii) the function `h` of the selected plan is executed by the Intention actor. After the execution of `v` is completed either by success or failure status, a message is sent to the actor which originally requested `v` containing the completion status and also a message is sent to Intention Pool actor signaling that this actor is now idle.

2.3.3 Translation Method

The translation method is designed to compile the models specified with the ASC2 DSL described in 2.3.1 into agents with the architecture described in 2.3.2. The full effective grammar description of ASC2 is presented in Listing 3.

[3] In the current implementation, the Intention pool actor exploits the *Router* feature of Akka.

```
agent             → initial_beliefs initial_goals plans
initial_beliefs         → (term '.')∗
initial_goals     → ('!' atomic_formula '.')∗
plans             → (plan '.')∗
plan              → ( '@' atomic_formula )∗
                  trigger_event ( ':' context )
                  '=>' body_definition '.'
trigger_event     → ( '+'|'-'|'+!'|'-!'|'+?' ) atomic_formula
context           → term
body_definition   → body_formula ( ';' body_formula )∗
body_formula      → ( '!'|'+'|'-' ) literal
                  |  loop
                  |  conditional
                  |  primitive_call
                  |  <VARIABLE> '=' term
term              → <VARIABLE>
                  |  '(' term ')'
                  |  <INTEGER> | <FLOAT> | <STRING> | 'true' | 'false'
                  |  <ATOM> '(' term_list ')'
                  |  term operator term
                  |  'not' term
                  |  '[' term_list ( '|' term )? ']'
                  |  <ATOM>
                  |  primitive_call
atomic_formula    → <ATOM>
                  |  <ATOM> '(' term_list ')'
literal           → <VARIABLE>
                  |  atomic_formula
term_list         → term ( ',' term )∗
operator          → '**'|'*'|'/'|'mod'|'+'|'-'|'='
                  |'=='|'!=='|'!='|'<'|'<='|'>'|'>='|'is'|'>>'
                  |'^'|'&&'|'||'|':-'
primitive_call    → '#' <ATOM> ( '.' <ATOM> )∗ '(' term_list? ')'
loop              → 'for' '(' <VARIABLE> 'in' term ')'
                  '{' body_definition '}'
conditional       → 'if' '(' term ')' '{' body_definition '}'
                  ('else' 'if' '(' term ')' '{' body_definition '}')∗
                  ('else' '{' body_definition '}')?
```

Listing 3: AgentScript's DSL grammar defenition

For each entity of the DSL, a mapping is defined to generate the code in the executable underlying language that can instantiate the objects with the desired semantics at run-time. The translated entities are then fitted in the abstract architecture to form an executable agent program.

Terms

The ASC2 DSL uses Prolog-style terms. In the translation of an script written in the DSL, each term (including inferential rules) maps to a `Term` object which encapsulates the parsed data (potentially containing nested `Terms`).

Any term in ASC2 can be translated and analysed in two ways: (1) external to the script by querying the belief base, and, (2) locally as part of the low-level code. The first approach makes use of any data-storage engine utilized by the belief base. This process is by default present in checking the context conditions of plans, but can also be used at any point in the agent's script; in our example the term:

```
restaurant(Loc) && not at(Loc)
```

can both be checked against the belief base to check if it can be proven, also a substitution for the variable `Loc` will be returned —that is if the belief base uses a prolog-like reasoning engine. In any case, this type of term utilizes the full capacity of the belief base which can also be less efficient. The translated Scala version of this term is illustrated in following, note that this is simply an object instantiation that can be analysed by the belief base's reasoning engine:

```
StructTerm(",",
    Seq(StructTerm("restaurant",
            Seq(vars("Loc"))),
        StructTerm("not",
            Seq(StructTerm("at",Seq(vars("Loc")))))))
```

The second type are terms that can be calculated locally, and for example are used in control flow structures `if/else` or variable assignments. As an example, if in the context of execution of a plan we have a variable `Loc` that is grounded by the string value `"french"`, the statement:

```
Loc + "_restaurant" == "french_restaurant"
```

can be calculated locally to boolean value `true` —e.g., in a variable assignment statement— without the need to query the belief base. Intuitively this second approach does not utilize the capabilities of the belief base but the local nature of its translation —it is calculated locally by the underlying language— can be very efficient. The translated Scala version of the second term is, not that unlike before this is already an term that can be calculated by JVM without the need for any extra reasoning[4]:

[4] Alongside some implicit type conversions

```
(vars("Loc") + StringTerm("_restaurant"))  ==
    StringTerm("french_restaurant"))
```

Access to the Lower-Level Language As consequence of an approach based on compilation, the DSL provides direct access to any object or function available in the agent's name space —in the Scala implementation, any object or function which is accessible via the Java class path. These lower-level access statements, indicated by the token `#`, are translated literally to the same statement in the underlying language. This capability provides fast and seamless reuse of libraries already established for the underlying language. Let us take the previous example, but this time also use a few of JVM's basic library methods, namely `String.join` and `String.toUpperCase`:

```
#String.join("_", Loc, "restaurant").toUpperCase == "FRENCH_RESTAURANT"
```

which translates to the valid Scala statement:

```
String.join(StringTerm("_"),
            vars("Loc"),
            StringTerm("restaurant"))
                .toUpperCase ==
                    StringTerm("FRENCH_RESTAURANT"))
```

Initial Beliefs/Goals and Inferential Rules

At syntactic level, initial beliefs and inferential rules are logic-style terms, and as such they translate to an `Term` object counterpart. Initial goals are a combination of a prefix (`!`, designating the adoption of a new goal) and a term and they translate to a `Goal` object encapsulating the prefix and a `Term` object. At initialization time, the initial beliefs are sent to the Belief Base actor to be asserted and initial goals are adopted sequentially in a synchronous manner, i.e., there is no concurrency.

Plan Rules

A plan rule $< e, c, h >$, should be translated into the object `{e,c,h}` which will be part of the plan library. The triggering event of the plan rule e consists of a trigger (one of `+!,-!,+?,+,-`) and a term t. The triggers convey the relevance of the plan to different event types while t can be seen as the payload of that event; `+!` relates to adoption of a new goal, `-!` relates to failure of a goal, `+?` relates to test goals, `+` and `-` respectively relate to assertion and retraction of a belief. The triggering event e then translates to an `Event` object which encapsulates the trigger and the translated `Term` object of t. The context condition c is a Term and translates to an `Term` object which can be sent to the Belief Base actor in a synchronous manner, and the response to that message determines if the plan is applicable in the current context and also returns the substitution for the variable

(if any). Following is an excerpt of a translation for an achievement goal plan with the signature `+!example_goal(ExampleParam)`:

```
1  object adopt_achievement_example_goal_1 extends IGoal {
2  ...
3      def execute(args: Arguments) = {
4          vars += ("ExampleParam" -> Argumenst(0))
5          val r0 = executionContext.beliefBase.query(context_condition)
6          if (r0.result == True) {
7              r0.bindings foreach { case (k, v) => vars(k).bind_to(v) }
8              plan0(vars)
```

where firstly (example, Line 4) the arguments of the event are added as local named variables based on the plan parameters, then (Line 5) the belief base of the agent is queried to check if the context condition of the plan is a logical consequence, and if that is true (Line 6) then all the variable substitutions of the query are added to local variables bindings (Line 7) and finally, the plan body is executed with the given variable bindings (Line 8).

The plan body h of a plan rule consist of zero or more steps. It is translated into a function, which contains the steps of h as imperative lines of code implemented in it. Each type of step is translated differently as is described below.

Primitive Actions A primitive action of the form `#h(...)` is in practice a lower-level function call in the underlying language and it is translated into a call to a function `h(...)` with its respective parameters. In the case of Java/Scala, this can be any callable entity on JVM's *classpath*. Take for example Scala's `println` method, to call this method as a primitive action a simple `#println("hello")` can be a statement used in ASC2, and a translation of this call will be:

```
primitive_handler.execute(() => println(StringTerm("hello")))
```

Note that in this translation the object `primitive_handler` is a dependency of the agent that in its default implementation simply calls the function passed to it, but, can be potentially customized and injected to the agent for more flexible control on what and how the functions are being called.

However, often the designer will need to define new domain specific methods to call from the agent's script as primitive actions, and these methods need access to the execution context of the agent itself. In every block of code in the translated code, there is an implicit value [5] called `executionContext` that contains the information about the agent and its execution context. This object contains information varying from simple attributes e.g., agent's name and type, to more

[5] This specifically refers to Scala's implicit values, these values do not need to be passed explicitly in a function call, instead if the corresponding method has declared them as an implicit parameter in its signature, Scala will automatically look for them and pass them along as a parameter.

technical information e.g., the current thread the function was called in, to even higher level information e.g., the intention or goal that was the source of the call or even on which agent's request this goal was adopted. To access this object from the context of a primitive action, the designer merely needs to declare it as an implicit parameter for the corresponding method. Take for example a simple primitive action that prints the agent's name and can be called from the script as `#print_name`, the implementation for this function will be:

```
def print_name()(implicit executionContext: ExecutionContext) =
    println(executionContext.name)
```

One of the most important advantage about how ASC2 handles primitive actions is that they do not need to be implemented in any specific class nor do they need to be passed to the agents at initialization time. This makes the development of agents and MAS much more scalable as primitive actions in a domain and the agent's scripts are completely isolated: the `#print_me` action can be built, packaged as a JVM artifact and placed in an artifact repository, then, it can simply be utilized by other designers by importing that artifact in a project[6].

Variables and Assignment Statements Variable assignment statements in form of `V = term` are used to (re-)assign the result value of an Term `term` to a variable `V`. ASC2 uses an internal map-like approach to store variables that also manages variable scopes, meaning that each code block (e.g, plan body, condition block) holds a map of all variables declared in that scope which also inherits the variables in its parent scope. Every read/write to a variable `V` then becomes a read/write operation to a member of the variable's map with the key `"V"`. A variable assignment is translated to an (overwriting) append operation for the variable's map by using the V as the key and `exp` as the value. As an example the statement `V = V + 1` is translated to:

```
vars += ("V" ->  (vars("V") + IntTerm(1)))
```

Belief Updates Belief update steps are composed of a prefix `+,-` and a term t. The prefixes respectively mean assertion and retraction. As the belief base of the agent is abstracted by the Belief Base actor, a belief update step is a blocking message to the Belief Base actor about assertion or retraction of the `Term` object of t. At a practical level, after translation belief-updates are technically low-level calls to a specific function, take for example the statement `+at(home)`, when translated becomes:

```
belief_handler.execute(
    Belief.ASSERT,
    StructTerm("at",Seq(AtomTerm("home"))))
```

[6]The more detailed exploration of development ecosystem in ASC2 is illustrated in Chapter 4

In this code the object `belief_handler` is a dependency of the agent that in its default implementation performs the aforementioned messaging with the Belief Base actor, but, can be potentially customized and injected into the agent for more flexible control on how belief updates are handled. Furthermore, ASC2 agents also react to belief-update triggers that may happen by assertion or retraction of a belief, however that process is completely controlled by the Belief Base actor that in its default implementation when a belief is asserted or retracted sends event messages to the Interface actor; this process can be customized by the designer[7].

Sub-Goal Adoption Task decomposition is crucial component of BDI-like agents and in essence is the ability to adopt sub-goals depending on the context of a plan. At the syntactic level, a (sub-)goal adoption is a prefix (e.g, `!,?`) plus a term t. The prefixes respectively mean achievement and test goals. In the translation method a sub-goal adoption step is translated as two phases, (i) a plan selection by using S_P is done to select and fetch a plan rule object `{e,c,h}` from the plan library, (ii) the function `h(...)` is called with any parameters that t may have as the arguments of `h`. This process becomes rather simple with the translation, as an example, the goal adoption step:

```
!example_goal(Var).
```

translates to the valid Scala statement:

```
adopt_achievement_example_goal_1.execute(Parameters(List( vars("Var"))))
```

which is effectively a simple JVM-level function call.

Control Flow Structures The compilation method of ASC2 supports a straightforward mapping of simple control flow structures such as loops and conditionals to their executable counterparts. The translation of these control structures to the underlying language is performed one-to-one; for example an `if/else` in the DSL is simply translated to an `if/else` in the underlying language. Take the simple statement:

```
if( Var =< 10 ) { Var = Var+1; }
```

based on the translation of terms and variable assignments, translates to the valid Scala statement:

```
if( vars("Var") <= IntTerm(10) ) {
    vars += ("Var" ->  (vars("Var") + IntTerm(1))
}
```

[7] At this point it may seem this much flexibility is not a requirement for a BDI framework. Evidently, most BDI frameworks are designed with the assumption of using a prolog-like backward-chaining query-response belief base. However, there are many interesting experiments that can be performed if that was not a limitation, an example illustrated in Chapter 5 where the belief base of the agents is replaced with a forward chaining norm reasoner that is continuously producing normative events for the agent.

2.3.4 Tools for Execution

The architecture of ASC2 agents is based on actors and for their execution these actors require an *actor system* that *spawn* and *start* them. The implementation of ASC2 is written in Scala and is based on the Akka framework. In addition to a compiler[8], it includes a minimal infrastructure that is able to spawn and start the compiled agents[9].

Communication Interface All the external communications of the agent are transmitted through one (or more) *transportation layer*. This layer needs to be specified to enable communication between agents. The framework remains agnostic with respect the transportation layer as long as there is an interface to convert messages from and to ASC2's message protocol. The default transportation layer makes use of Akka's typed messages. However, this layer is one of the dependencies that is injected into the agents at initialization time. This allows the designer to implement this layer with any technology (e.g., REST API, gRPC, Message Queues) and inject it into agents to allow them to communicate with other agents (and other entities) using that technology without the need to modify the framework or even the script of the agents.

The communications interface of the agents is based on speech act preformatives. On a practical level, they are implemented with actions like `#achieve` which relays an achievement goal event, `#tell` and `#untell` which relay belief-update events, and `#ask`/`#respond` which can be used between agents as synchronous communication with test goal events.

2.4 Performance Analysis

The following section proposes quantitative comparisons between the ASC2 framework and three other frameworks: Jason (`v2.5`), ASTRA (`v1.0.0`) and Sarl (`v0.11.0`). Jason [26] was chosen because, like ASC2, it uses a language based on AgentSpeak(L), is implemented in Java and as reported by [36] potentially outperforms other BDI frameworks. ASTRA and Sarl are both also implemented in Java, but, more importantly, like ASC2, rely on a compilation approach.

Performance comparison is effectuated by means of two fairly standard benchmarks (token ring, chameneos redux), close to what has been presented in [36]. The main difference w.r.t. [36] is the metrics, as we separate the interpretation/setup time from the execution time, to present better insights on how these frameworks operate. An additional benchmark (service point) was also performed to assess the ability of the frameworks to allow concurrent decomposition of tasks inside the agents. The benchmarks were performed on a Debian GNU/Linux 10 machine

[8]Source code: `https://github.com/mostafamohajeri/scriptcc-translator`.
[9]Source code: `https://github.com/mostafamohajeri/agentscript`.

with an 8 core Intel(R) Xeon(R) CPU E5-1620 v4 @ 3.50GHz CPU and 64GB of RAM using Java version 11 with GraalVM 20 JRE. Each benchmark was performed 10 times and the JVM was stopped between each run to avoid the impact of one experiment on the next.

In the first two benchmark scenarios, three metrics are recorded: (1) total interpretation/setup time, including agent creation time, (2) internal execution time measured from the instant that the first agent starts until the completion of the test, and (3) CPU core load. Execution and data gathering is controlled by a Python script that runs the benchmarks in the desired dimensions and records the metrics[10].

2.4.1 Token Ring

The token ring benchmark is a simple program targeting multiple aspects of parallel frameworks: handling different number of agents, message passing and level of concurrency each agent can achieve. The testing scenario consists of W worker agents, T tokens are distributed among the workers, and each token has to be passed N times in a ring. When all T tokens have been passed N times, the program ends. To run this benchmark a program should:

- create W number of workers;
- each worker should be connected to its neighbor forming a complete ring;
- initially each token $1 \leq i \leq T$ is assigned to a worker $1 \leq j \leq W$ with the equation $j = i * (W/T)$
- each worker sends the token to its neighbor
- The program finishes when all T tokens have been passed N times

The script for a worker agent in ASC2 is presented in Listing 4. The experiment was performed by varying W, T and N independently within the values $\{4, 16, 256, 1k, 4k\}$, resulting in 125 different configurations for each framework. We also put a 1 minute limit for each execution and anything beyond that is considered a *timeout*.

Implementation Notes

In all implementations a *broker* agent is present that starts the benchmark by distributing the tokens and gathers the completed tokens to stop the execution. There is a difference in the Sarl implementation. As Sarl does not provide a central agent resolver to address agents by name, an extra step is implemented in the broker to iterate over all worker agents and link them together in a ring.

[10]Source code: `https://github.com/uva-cci/aop-benchmarks-agere2020`.

```
+!init(W) : W > 1 =>
    Nbr = "worker" +
        ((#asString(Name).replaceAll("worker","").toInt mod W) + 1);
    +neighbor(Nbr).

+!token(0) =>
    #achieve("master", done).

+!token(N) : neighbor(Nbr) =>
    #achieve(Nbr, token(N - 1)).
```

Listing 4: Token ring `worker` script in AgentScript DSL

Results

A summary of the results for this benchmark is illustrated in Figures 2.3 and 2.4. In Figure 2.3, the number of agents W is the variable while N and T are kept constant with two settings ($N = 256, T = 256$) and ($N = 4k, T = 4k$). Only Jason and ASC2 were able to execute ($N = 4k, T = 4k$). Sarl was able to only execute the benchmark up to $W = 256$ agents and timed out with a warning[11].

ASTRA seemed stable enough to finish the ($N = 4k, T = 4k$) test but not within 1 minute. ASTRA executes very poorly for ($N = 256, T = 256$) test, especially with lower number of worker agents, plausibly because with less worker agents each agent has more concurrent threads of work to execute. ASC2 and Jason both perform almost without much effect w.r.t. number of agents, suggesting that both frameworks can handle concurrency inside agents to a good extent, although in all cases Jason performs marginally better.

In Figure 2.4 another view on the results is presented. This time the variable is the number of tokens T, whereas W, N are kept constant in two settings: ($W = 256, N = 256$) and ($W = 4k, N = 4k$). Like in the previous results Sarl could only finish the ($W = 256, N = 256$) test. ASTRA was able to execute the ($W = 4k, N = 4k$) test but only up to $T = 1k$ and timed out after that. In the ($W = 256, N = 256$) Jason and ASC2 performed much better and scaled almost linearly with the number of tokens which shows that both frameworks can handle the increased concurrency and the higher number of messages to be passed in an efficient manner. On the other hand Sarl and ASTRA performed poorly under the increasing amount of tokens. In the ($W = 4k, N = 4k$) test Jason performs marginally better than ASC2.

[11] `Potentially dangerous stack overflow in locks.ReentrantReadWriteLock`. We suspect this occurs because at the start all workers need to send a message to the broker to get their neighbors and the broker can not handle this amount (≥ 1024) of concurrent messages.

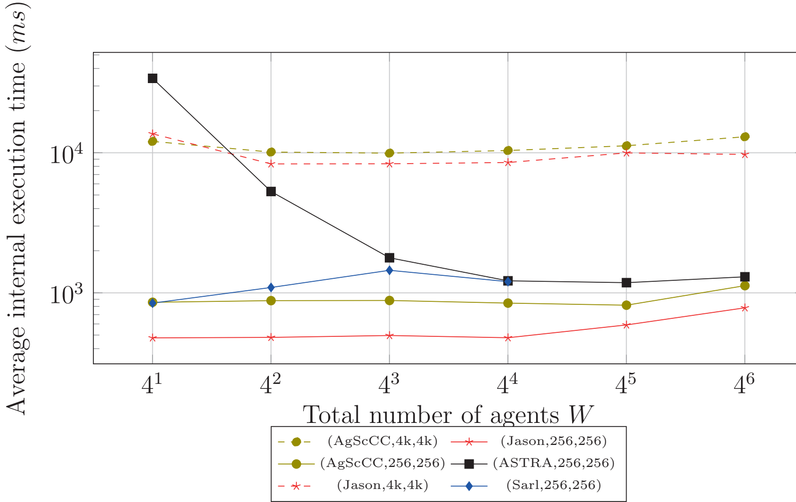

Figure 2.3: Token ring results for each (framework, T, N)

CPU Load Figure 2.5 and Figure 2.6 present the average core load during the token ring test respectively in the $W, T = 256$ and $N = 4096$ and in the $W, T, N = 4k$ settings. In the lower settings (Figure 2.5) Jason and ASTRA have much less CPU demand than ASC2 and Sarl. On the other hand, in the higher setting (Figure 2.6) the CPU load between Jason and ASC2 is closer (respectively 85.7% and 88.6%, vs 57.7% and 77.7% in the lower setting). This can be an indication that ASC2 has a higher footprint on the CPU load, especially for initialization time.

To understand how much each framework can distribute the load between CPU cores we have to look at the standard deviation of CPU load data. A higher deviation indicates that the framework is not balancing the load between cores. ASTRA shows to have very poor load balancing with the deviation almost as high as the average which can mean that some of the cores are not even used in execution. Sarl has a high balancing of cores even in lower setting. In the higher settings both Jason and ASC2 seem to distribute the load between CPU cores sufficiently.

Initialization Time To assess the initialization time, total execution time is subtracted by the internal execution time in the lowest setting with $N = 4k$ and $T = 4k$ and the results are presented for an increasing number of agents in Figure 2.7. ASTRA proves to have the fastest initialization, at least up to $4k$ agents, followed by Jason and closely by ASC2. Sarl seems to have the slowest initialization time and scales very badly with the number of agents.

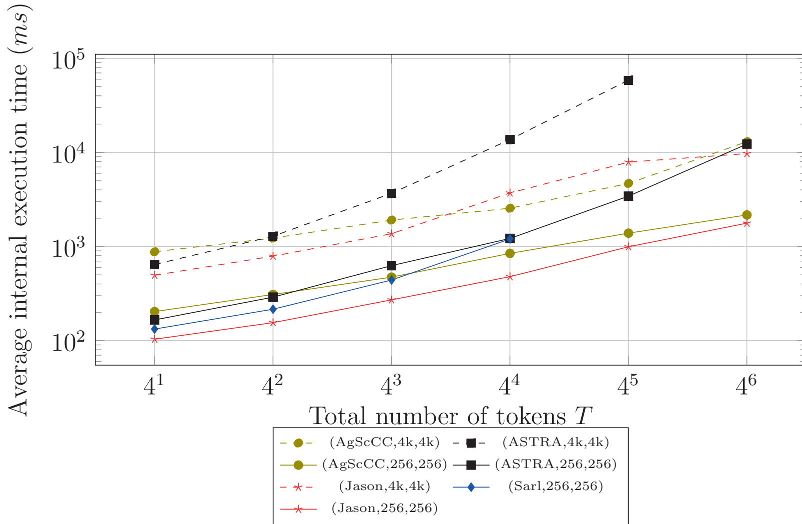

Figure 2.4: Token ring results for each (framework,W, N)

2.4.2 Chameneos Redux

The second benchmark is adopted from [104] and is a test intended to capture the effects of one limiting point to the execution framework. The scenario consists of C chameneo creatures living in the jungle; they can go to a common place to meet other creatures and *mutate* with them. Each creature has a color assigned to it from a color pool and after mutation its colour changes based on the color of the other creature it met. These meetings should happen for a total number of N times. To run this benchmark a program should:

- create C differently colored (blue, red, yellow), differently named, concurrent chameneo creatures
- write all the possible complementary color combinations;
- write the initial color of each creature;
- each creature will repeatedly go to the meeting place and meet, or wait to meet, another chameneo;
- both creatures will change color to complement the color of the chameneo that they met;
- after N meetings have taken place, for each creature write the number of creatures met and the number of times the creature met a creature with the same name (should be zero).

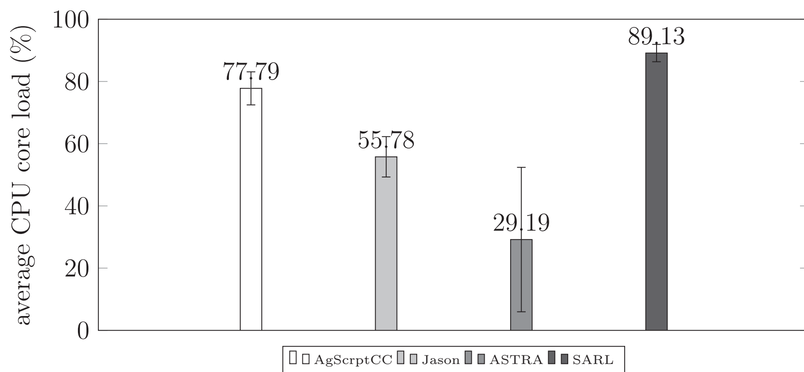

Figure 2.5: CPU load (average and standard deviation on 8 cores) in token ring with $N = 4k$, $T = 256$ and $W = 256$

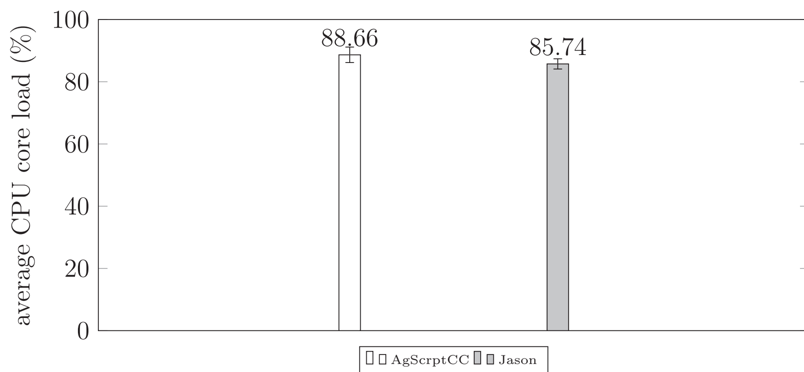

Figure 2.6: CPU load (average and standard deviation on 8 cores) in token ring with $N = 4k$, $T = 4k$ and $W = 4k$

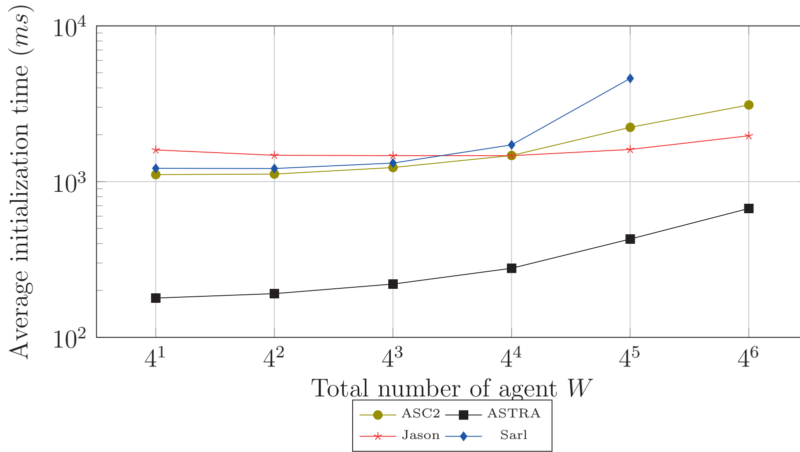

Figure 2.7: Initialization time in token ring with $T = 4$, $N = 4$

- the program finishes when N meetings have happened.

The experiment was performed with the set of variables $C = \{64, 256, 1k, 4k\}$ and $N = \{1k, 4k, 16k, 64k\}$. This provide us with 20 different configurations for each framework. All tests were given a 1 minute time limit and it is considered a timeout after that.

Implementation Notes

In all implementations a *broker* agent is present that acts as the meeting point for chameneos. This agent is the main point of this benchmark as it will be constantly under high number of requests from the chameneos agents.

Results

The first view on the results is presented in Figure 2.8. In this setting the number of meetings N is kept constant at two values $4k$ and $64k$ whilst the number of chameneos is the variable. The results show that Jason and ASC2 scale well with the number of agents while ASC2 performs marginally better in the $N = 64k$ test. Sarl and ASTRA suffer from the higher number of agents to the point that Sarl could finish both tests only up to $C = 1k$ agents while ASTRA finishing $N = 64k$ test only in the $C = 64$ agents setting.

Figure 2.9 presents another view on the results. This time the number of chameneos C is kept constant at 256 and $4k$, whilst the number of meetings N is the variable. Sarl could only finish the $C = 256$ test while ASTRA could only finish it up to $N = 16k$ and timing out after that. ASTRA was also only able to

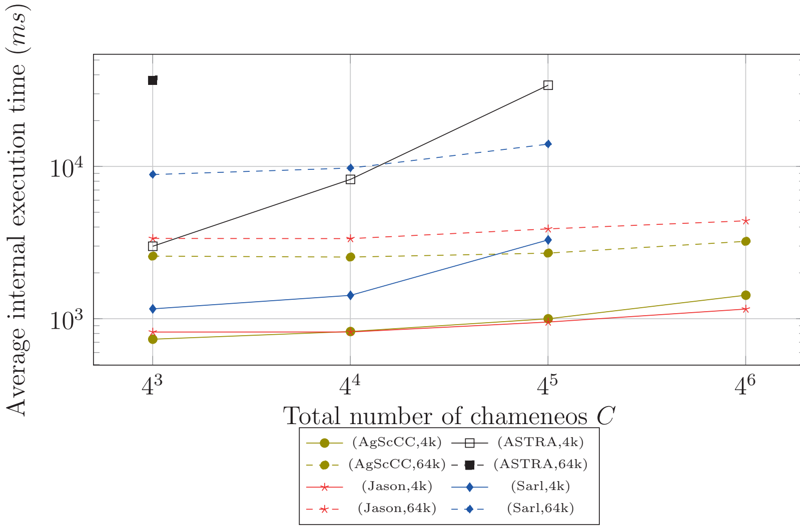

Figure 2.8: Chameneos redux results for each (framework, N)

finish the $C = 4k$ test with $C = 64$ number chameneos. ASC2 and Jason both completed the tests with linear scaling, with ASC2 outperforming Jason slightly in the $C = 4k$ test. This shows that both Jason and ASC2 can handle higher levels of concurrency in the broker agent w.r.t. the increasing number of concurrent requests.

2.4.3 Service Point

This last benchmark is not about performance. Rather, it is designed to illustrate the differences between the execution in a *step-based* framework like Jason in contrast to a compilation-based framework like ASC2, focusing on how they handle actions (namely time-consuming primitive actions) specified outside their DSL. The scenario of this benchmark consists of one service point and N number of consumers. Each consumer sends R requests to the service point and waits for the response. The service point needs a random amount of time t ($0 \leq t \leq 5000$ ms) to process each request. A simple `Thread.sleep(t)` is used to mimic thread time consumption. To run this benchmark a program should

- create 1 service point and N service consumers.
- each consumer will send R number of requests to the service point
- the program finishes when all of the $R * N$ requests have been responded

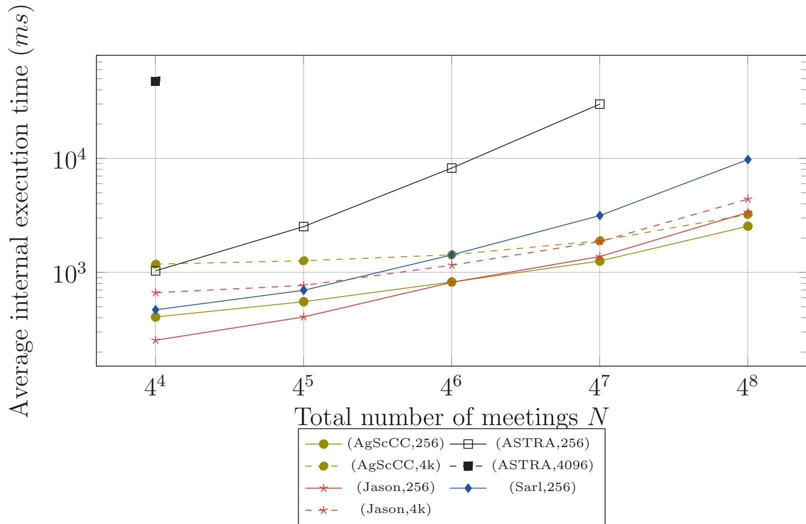

Figure 2.9: Chameneos redux results for each (framework,C)

The experiment was done only on Jason and ASC2 with variables $N = \{1, 4, 16\}$ and $R = \{1, 4, 16\}$. With respect to total number of request $R * N$, this gives us with 5 unique configurations. To account for the non-determinism added by the randomization each configuration is executed for 100 times.

Results

The results of this experiment are presented in Figure 2.10. Jason performs much worse in this scenario, as it is not being able to finish the 256 requests within a 200 seconds timeout. This is even more strange as in our setting Jason is set to use 8 threads and ASC2 to 6 and by looking at the results we can see that ASC2 is always using the thread times completely but Jason is not. The reason for this is that Jason uses a *sequential* reasoning cycle inside each agent; at every reasoning cycle, a Jason agent takes the next step from each of its intentions and executes them. The reasoning cycle ends when all intentions execute one step. This means that if in the reasoning cycle of an agent one of these steps is a time-consuming primitive action, the whole cycle will be blocked[12]. On the contrary a compiled agent does not have any notion of steps at run-time and the parallelism between intentions of the agent is also handled by the underlying concurrency model, in this case the Actor model. This matter is further discussed in 2.5.3.

[12]Jason provides extra built-in directives like `.wait` to mimic unblocking suspension of intentions but that is beyond the context of this benchmark.

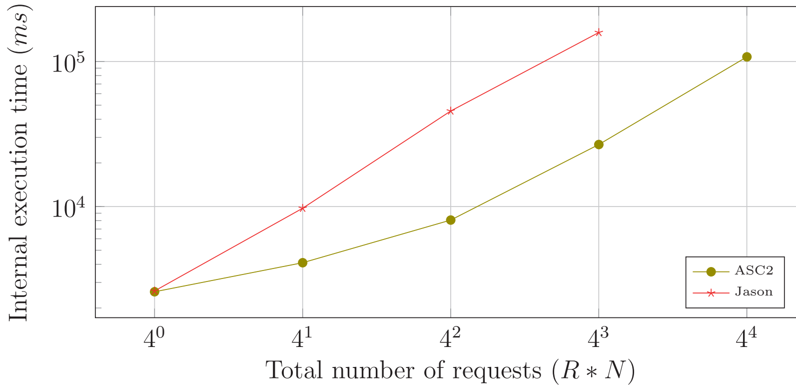

Figure 2.10: Service point scenario results

2.5 Discussion

This chapter presents and evaluates a framework for an AOP language based on AgentSpeack(L) relying on compilation. Compilation in this context is not novel as it has been used previously by other AOP frameworks like SARL [141] and ASTRA [65]. The novelty of this work lies in two aspects. First, unlike SARL and ASTRA, that use a DSL very close to their underlying language (Java), ASC2 uses a logic-based DSL close to AgentSpeak(L). As our pipeline starts from an *antlr* grammar, in principle the current DSL can be replaced by any other AOP language that can be mapped to the ASC2 abstract execution architecture. Second, our approach maps the DSL into an architecture that exploits the Actor model. This means that not only the final executable model is more robust, because it takes advantage of the established concurrency model and the maturity of the libraries implementing the Actor model (e.g, Akka), but also that the translation itself is an open process, so its product becomes in principle more understandable for the programmer.

2.5.1 Performance

The execution model of ASC2 is closer to Sarl and ASTRA than to Jason (see 2.5.3), but, as shown by the benchmarks, it is substantially outperforming both Sarl and ASTRA. At the same time, ASC2 performance was below what we expected before running these experiments. Investigating possible causes by profiling the execution of benchmarks, we found out that a considerable amount of execution time is spent on the blocking due to *synchronized query calls* to the belief base. These calls had to be synchronized because Prolog engines like *Styla* and *tuProlog* [64] (another candidate solution we tested for handling belief bases)

are inherently made for single thread access. Even a simple *read* query to a Prolog engine still counts as *write* access because of the backtracking. We believe once this issue is addressed the performances of ASC2 will greatly improve.

2.5.2 Language

Although all of the considered frameworks propose DSLs to program reactive agents, their bases are different. Agent-ScriptCC's DSL is based on AgentSpeak(L), which gives to the language a logic-oriented flavour; this is also the case for Jason, and both frameworks can take advantage of Prolog-style terms. ASTRA's DSL is also based on concepts defined in AgentSpeak(L) but with more syntactic resemblance to Java. Sarl's language does not try to be a logic-based language, therefore it does not contain components corresponding to terms; it is rather very close to Java.

2.5.3 Execution and Parallelism

As a common ground, all these frameworks are used to specify reactive agents, but they differ in how the agent's (re)actions are executed. The most particular solution comes with Jason which uses the concept of *steps*. The Jason interpreter treats each symbolic step/instruction in a plan of a reactive rule as a single unit of execution, and emulates an imperative program by executing them in a sequence in consecutive reasoning cycles. In contrast, in the other three frameworks, the steps of the reactive rules are already imperative programs ready to be executed. The approach taken by Jason has important consequences especially when agents execute multiple parallel threads of work (intentions) at the same time. This concept is examined more in detail in section 2.4 and in [65].

2.5.4 Access to the Lower-Level Language

One of the motivations behind developing ASC2 has been to enable access to libraries defined in the underlying general-purpose programming language of choice in a easy and seamless way. In our view this impacts the usability of the framework in larger applications. Leading by an example, consider a programmer that needs to call the Java function `Thread.sleep(1)` in a reactive rule. In Jason one needs to create an extra class extending one of Jason's internal classes (`Agent`, `Action` or `Environment`) and define a method that wraps this low level function and then call the wrapping method from the agent program. In ASTRA it is almost the same as Jason and one needs to create a class extending the type `Module`, wrap this function inside a method, and annotate it appropriately to be able to call it from the agent program. On the opposite side, this is entirely different for Sarl and ASC2, as one can simply call this function directly from the agent program. In case of ASC2 this can be done with `#Thread.sleep(1)`.

2.5.5 Communication

The communication in ASC2 is entirely externalized, both for agent-to-agent and agent-to-environment communication. In the current implementation communication between agents uses Akka's internal message system but this can easily be replaced with any other type of communication mechanism, e.g. by using a message queue (MQ) to be able to execute the agents in a distributed setting. For the other frameworks, externalization is possible, but requires specific wrappers to the communication infrastructures (Jason with JADE).

2.6 Conclusion

The slowly but steadily increasing interest in programming languages based on BDI or functionally similar architectures for virtual assistants, robotics, (serious) gaming, as well as for social simulations, hints that there is a general consensus that these solutions might be suitable to reproduce human-like reasoning, or rather human-intelligible computation.

Historically, the majority of contributions in this area were concerned mostly by the logical aspects of the problem rather than its computational aspects [96]. However, more recent contributions revealed the presence of issues w.r.t. computational performance and compatibility to modern environments and tools, motivating efforts to redevelop existing BDI frameworks according to best practices [6, 3]. Looking at intentional programming in the longer term, we need to acknowledge that operational settings differ from the typical low-scale simulation setting in which it is used today. Besides a difference in scale, components can also be fully distributed. Because of this, a future target feature of ASC2 will be the capability to deploy and execute agents in distributed settings. This seems to be an achievable objective as there are already approaches available to run actors in distributed environments.

An initial, additional motivation of using an actor-oriented architecture for the intentional agents is that by having this extra level of abstraction the agent become more modular, enabling the augmentation of agents with complementary machinery like using AI modules [152], normative reasoning modules [120], planning (e.g. HTN, STRIPS) modules [119] and preference checking modules [162, 122]. Defining adequate interfaces to support the different types of add-ons for ASC2 agents will be investigated in the future.

Finally, the benchmarks reported in this chapter demonstrates that despite the initial maturity level of the framework, ASC2 is already competitive against existing frameworks, motivating further exploration.

Chapter 3

Transparent Decisions in Social Actors: Preferences

Computational agents based on the BDI framework typically rely on abstract plans and plan refinement to reach a degree of autonomy in dynamic environments: agents are provided with the ability to select *how-to* achieve their goals by choosing from a set of options. In this chapter we focus on a related, yet under-studied feature: *abstract goals*. These constructs refer to the ability of agents to adopt goals that are not fully grounded at the moment of invocation, refining them only when and where needed: the ability to select *what-to* (concretely) achieve at run-time. We present a preference-based approach to goal refinement, defining preferences based on extended *Ceteris Paribus* Networks (CP-Nets) for an AgentSpeak(L)-like agent programming language, and mapping the established CP-Nets logic and algorithms to guide the goal refinement step. As a technical contribution, we present an implementation of this method that solely uses a Prolog-like inference engine of the agent's belief-base to reason about preferences, thus minimally affecting the decision-making mechanisms hard-coded in the agent framework. The aim of this chapter is (1) to introduce a generic approach for embedding explicit preferences into BDI agents, and (2) to provide a proof-of-concept implementation of this approach in ASC2.[1]

3.1 Introduction

Computational agents based on the BDI framework typically rely on abstract plans and plan refinement to reach a degree of autonomy in dynamic environments. In practice, relative autonomy in this context consists in the ability of an agent to select *how-to* achieve their goals by choosing from a set of options. BDI agent scripts typically consist of hierarchical, partial, abstract plans. This contrasts with

[1]The material presented in this chapter refines and extends elements presented in [122, 123, 124].

classic forms of planning, providing agents with fully-grounded policy, designed to reach a certain specific objective.

This work focuses on a related, yet under-studied feature: *abstract goals*. These constructs refer to the ability of agents to adopt goals that are not fully grounded at the moment of invocation, refining them only when and where needed, that is, the ability to select *what-to* (concretely) achieve at run-time. Examples of abstract goals can be found typically in *activity*-level characterizations of behaviour, e.g. walking (where?), eating something (what?), meeting someone (who?), selling (what? to whom?), etc.

The specification of abstract goals is a feature already present in some agent frameworks as those based on the AgentSpeak(L) language, albeit they rely on simplistic mechanisms for goal refinement. The present work aims to cover the goal refinement step (from abstract goals to concrete goals) as part of the agent's decision-making cycle. For doing so, we present a preference-based approach to goal refinement. We start from defining preferences based on *Ceteris Paribus* Networks (CP-Nets) [29]—more precisely, in the extended form of *Ceteris Paribus* Theories (CP-Theories) [166]—and we consider the established CP-Net logic and algorithms to guide the goal refinement of the agent. At implementation level, our target is an AgentSpeak(L)-like [139] agent programming language. Since Jason [26], AgentSpeak(L) programs are enriched with Prolog rules and facts for knowledge-level processing, occurring e.g. for testing context conditions during the plan selection phase. We present therefore an implementation of a preference-based goal refinement method that solely uses a Prolog-like inference engine of the agent's belief-base to reason about preferences, requiring only a minimal modification to the decision-making mechanisms hard-coded in the agent framework. To achieve this, a transformation method is proposed to map an extended version of CP-Nets and CP-Theories into Prolog facts and rules for the script of the AgentSpeak(L) agent, as well as a Prolog implementation of the algorithms necessary to reason with preferences.

The chapter proceeds as follows: section 3.2 provides a background about the concepts used in this work; section 3.3 presents the method and examples for preference-based abstract goal refinement in AgentSpeak(L) agents; section 3.4 describes a practical implementation of this method, and section 3.5 elaborates a discussion and conclusion over the proposed method.

3.2 Background

3.2.1 Abstract Plans in BDI Agents

Agents specified following the BDI paradigm are characterized by three mental attitudes. Beliefs are facts that the agent believes to be true. Desires capture the motivational dimension of the agent, typically conflated with the more concrete

form of *goals*, representing procedures/states that the agent wants to perform/achieve. Intentions are selected conducts (or *plans*) that the agent commits to (in order to advance its desires).

Since their origin [140], the essential feature associated to BDI architectures is the ability to instantiate abstract plans that can (a) react to specific situations, and (b) be invoked based on their purpose. Consequently, the BDI execution model often relies on a *reactive* model of computation, usually in the form of some type of *event-condition-action* (ECA) rules often referred to as *plans*. Plans are uninstantiated specifications of the *means* (in terms of course of actions) for achieving a certain *goal* [140]. These constructs represent the procedural knowledge (*how-to*) of the agent. There are multiple proposals in the literature for programming language and architecture of BDI agents, the most commonly used being AgentSpeak(L) [139], which will serve as basis for the present proposal.

3.2.2 Preference Languages

Preferences play a crucial role in decision-making [135]. Several models of preferences have been presented in the literature (e.g. on decision-making, planning, etc.), with various levels of granularity and expressiveness (see e.g. [70]). Several models of preferences have been presented in the computational literature, with various levels of granularity and expressiveness. For a comprehensive overview (specifically in AI planning), we direct the reader to [102, 30]. On a higher level, preference representation methods can be divided into *quantitative* and *qualitative* [102].

The most straightforward *quantitative* approaches are based upon *utility theory* and related forms of decision theory.Typically, in quantitative approaches there is a utility function that assigns to each action in each state a (negative or positive) value, then the agent/the planner system tries to *maximise* its utility by choosing actions that would result in higher total utility (including avoiding actions with negative utility, e.g. due to cost).

A hybrid quantitative method is provided by PDDL3 [83], an extension of the *planning domain definition language* (PDDL) [118]. Although based on qualitative descriptions, these preferences are considered quantitative [7] because the valuation of each preference is expressed with a numerical value. Although utility-based approaches bring clear computational advantages, they also suffer from the non-trivial issue of translating users' preferences into utility functions.

This explains the existence of a family of *qualitative* or hybrid solutions. The *logical preference description* (LPD) language [32] uses ranked knowledge bases alongside preference strategies to present preference descriptions. The LPP language [16] is a first-order preference language defined in situation calculus to reason about conditional and qualitative preference. Other preference models, such as GAI networks [88], CP-Nets [29] and qualitative preference systems (QPS) [163], have been specifically introduced for taking into account dependencies and

conditions between preferences. GAI networks build upon the assumption of *generalized additive independence*, and in doing so they enable computing the utility contribution of every single attribute/subset of attributes. QPS offers a framework for representing multi-criteria preferences based on a lexicographic rule which combines basic preferences over variables, and a cardinality-based rule which counts criteria that are satisfied; QPS have been extended with goal-based preferences [164], allowing to define preferences from the context of goals.

In the present work, we decided to focus on CP-Nets, and their extension CP-Theories [166], for two main reasons: they rely on weaker assumptions, and exhibit primarily a qualitative nature.

Ceteris Paribus networks (CP-Nets)

Conditional *ceteris paribus* preferences networks (CP-Nets) are a compact representation of preferences in domains with finite *attributes of interest* [29]. An attribute of interest is an attribute in the world (e.g. *restaurant*) that the agent has some sort of preference over its possible values (e.g. *italian* and *french*). CP-Nets build upon the idea that most of the preferences people make explicit are expressed jointly with an implicit *ceteris paribus* ("all things being equal") assumption. For instance, when someone says "I prefer a French restaurant over an Italian one", they do not mean at all costs and situations, but that they prefer a French restaurant (over an Italian one), all other things being equal. An example of *conditional preference* is "If I'm at a French restaurant, I prefer fish over meat". CP-Theories [166] extend CP-Nets adding stronger conditional statements with the construct "*regardless of*", allowing some attributes to be released from the equality rule.

In general, CP-Nets can be associated with two tasks: (1) finding the most preferred outcome on a certain domain of variables (2) comparing two outcomes with different criteria. Both CP-Nets and CP-Theories provide efficient algorithms for these tasks.

Preferences in BDI Agents

Goals are used to identify desired states or outcomes, and preferences are used to identify more (or less) desired states or outcomes. While goals are a central aspect in BDI agents, so far, none of the main BDI frameworks and languages include preferences as part of the definition of the agents. This explains why there exist already previous studies that, like this work, attempt to enhance BDI agents with explicit preferences. Visser et al. [161, 162] present an approach to embed preferences defined in the LPP language into BDI agents to guide plan selection. Nodes in the goal-plan tree of the agent are annotated by the designer about the effects of that plan and then this information is propagated automatically to other nodes in the tree at compile time. Then, at run-time, the

agent uses the LPP logic to select the most preferred plan for a goal based on this information. Dasgupta et al. [53] proposes a lookahead method to enhance AgentSpeak(L) agents with constraints and objectives that the agent can use for plan selection at run-time; their approach also requires annotations for plans to reason about the preferability of plans. Mohajeri et al. [122, 123] add preferences in form of CP-Nets in AgentSpeak(L)-like agents, however, their approach consider a cross-compilation step: they annotate primitive actions of agents with their expected effects, and this information is then propagated through the goal plan tree to create a conditional ordering between plans. Padgham et al. [131] add situational preferences as part of plan definitions in a BDI language. The agent can use them to quantify the value of each plan at run-time. Their method is similar to this work in the sense that it does not require any lookahead, but it is different because they add preference valuations as part of each plan, which are then used in plan selection with the implicit preference of maximizing them—this makes the approach essentially a quantitative one.

3.3 Method

3.3.1 AgentSpeak(L) Agents

While this work mainly focuses on ASC2 as the BDI framework, the methods proposed in this chapter target any frameworks that utilize AgentSpeak(L). The main point of interest in BDI reasoning cycle for the preference reasoning approach proposed in this chapter is the plan instantiation process. This process, as it was briefly introduced in the previous section, starts when an event is selected for processing. Firstly the event is unified with the triggering events of the plans in the plan library. The ones that do unify are called relevant plans and the resulting unifier is called the relevant unifiers. Then, for each relevant plan, the relevant unifier is applied to its context condition and the result is queried against the belief base to create zero or more substitutions such that the context is a logical consequence of agent's current belief. The composition of relevant unifier with each substitution is called an *applicable unifier*. The following definitions apply:

Definition 3.1 (Plan)
A (reactive) plan is specified by $e : C \Rightarrow H$ *where* e *is a triggering event,* C *is a formula capturing context conditions, and* H *is a sequence of sub-goals or actions to be performed at the occurrence of the trigger event.*

Definition 3.2 (Relevant plan)
A plan in the form of $e : C \Rightarrow H$ *is a* relevant plan *with respect to an event* ϵ *iff there exists a most general unifier* σ *such that* $\epsilon\sigma = e\sigma$*. Then,* σ *is referred to as the* relevant unifier *for* ϵ*.*

```
1  % (P1)
2  +!go_order(Loc,Meal) :
3      restaurant(Loc) & not at(Loc) =>
4          #move_to(Loc);
5          !order(Meal).
6  % (P2)
7  +!go_order(Loc,Meal) :
8      restaurant(Loc) & at(Loc) =>
9          !order(Meal).
10 % (P3)
11 +!order(meal(S,M,W)) :
12     meal(S,M,W) =>
13         #ask_waiter(meal(S,M,W)).
```

Listing 5: Reactive Plans of Food-ordering Agent

Definition 3.3 (Applicable plan)
A plan in the form of $e : C \Rightarrow H$ *is an* applicable plan *with respect to an event* ϵ *iff there exists a relevant unifier* σ *for* ϵ *and there exists a substitution* δ *such that* $C\sigma\delta$ *is a logical consequence of belief base* B*. The composition* $\sigma\delta$ *is referred to as the* applicable unifier *for* ϵ *and* δ *is referred to as a* correct answer substitution*.*

Example 1

Imagine again the domestic robot from the previous chapter, an agent that upon request, can go to a restaurant and order a three-course meal. This time the agent's plans and beliefs are expanded to give it more choice. The script for such agent is presented in Listing 5. The agent has two plans for going to a restaurant and ordering a meal: the first plan (P1) is *applicable* if the agent is not at a restaurant at the moment, which means a step of moving (`#move_to` primitive action) is needed prior to ordering the meal; the second plan (P2) is applicable if the agent is already at the restaurant which means the agent will just adopt the goal of ordering the meal. There is also one plan for ordering the meal (P3) which is applicable if the agent has the belief that the meal it wants to order exists. Suppose the agent selects an event with the trigger:

```
!go_order(french,meal(veg,meat,white))
```

For this event, both plans (P1) and (P2) are relevant with unifier σ:

```
{Loc/french, Meal/meal(veg,meat,white)}
```

Assuming that the belief base of the agent contains the beliefs `restaurant(french)` (meaning that there exists a French restaurant) and `at(home)` (meaning that the agent is at home), then only the first plan will be an applicable plan for this event,

and the applicable unifier will be the same as the relevant unifier. This entails that only plan (P1) will be instantiated as:

```
+!go_order(french,meal(veg,meat,white)) :
    restaurant(french) & not at(french) =>
    #move_to(french);
    !order(meal(veg,meat,white)).
```

Note that in case the agent had more than one applicable unifiers, meaning it had more than one option to react to this goal, then the S_O function would have been called to select one of the options.

3.3.2 Abstract Events, Abstract Goals

Partial autonomy in dynamic environments is considered a core attribute of BDI agents, and this is in fact one of the reasons that separates plan refinement in BDI agents from classical planning approaches [57]. While the idea of choosing between distinct plans to achieve a certain goal—typically referred to as plan selection—has been investigated by the community as the principal point of autonomous choice in BDI agents, there is indeed another important type of autonomy embedded in BDI agents: abstract events. While the previous example only exhibited fully grounded events, BDI agents, in particular those derived from AgentSpeak(L) [140, 139] can indeed handle abstract events, referring to situations where an event contains unbounded variables and these variables can be grounded by different means such as context conditions of plans or test goals at any level in the plan refinement of the event. It can be argued that if plan selection promotes autonomy in the *how-to* dimension of the agent, abstract events, including the invocation of abstract goals, promote autonomy in selecting (concretely) *what-to* with it.

Example 2

Consider the same agent presented in listing 5. This time we assume the agent has more information about the environment: it has beliefs about two types of soups, two types of main course, two types of wine, two restaurants, also it believes that it is standing already in one of the restaurants (the `french` one), and finally it has an inferential rule for which all possible triple of soup, main course and wine form a meal combination. Those beliefs are presented as in listing 6.

Now assume the agent receives an abstract event `!go_order(L,M)` which basically puts no constraints over where the agent should go and what it should order, and so gives it full autonomy to choose how to proceed. When the agent receives this event, both plans P1 and P2 are considered relevant plans with unifier `{Loc/L, Meal/M}`. But considering the context conditions and the belief base, P1 will be applicable with unifier `{Loc/italian, Meal/M}` and P2 with `{Loc/french, Meal/M}` (note that in both cases the second parameter is not

```
main(fish). main(meat). soup(veg). soup(fish).
wine(white). wine(red).
restaurant(french). restaurant(italian).
at(french).
meal(S,M,W) :- soup(S), main(M), wine(W).
```

Listing 6: Beliefs of Food-ordering Agent

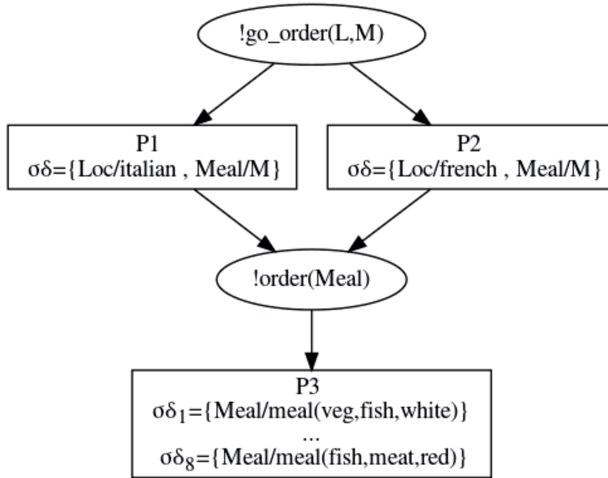

Figure 3.1: Goal-Plan Refinement of the Agent

grounded as it is unified to another variable). At this point the agent's reasoning engine needs to use its plan selection function to choose one of the two plans. In both cases, the next event for the agent will be `!order(M)` and P3 is a relevant plan for this event with unifier `{M/meal(S,M,W)}`. Taking into account the unification occurring at context conditions, this event will have in principle $2^3 = 8$ different applicable unifiers with all possible combinations for the meal, e.g: `{M/meal(veg,fish,white)}`, for which, again, the plan selection function needs to choose an option to start the actual execution. The goal-plan tree of this abstract goal can be seen in Figure 3.1.

In current implementations of BDI frameworks based on AgentSpeak(L), the three selection functions S_E, S_O, S_I (respectively for events, plans/options, and intentions) are typically exposed as abstract functions that the designer can override to implement any type selection function. Although this approach promotes flexibility, the fact that part of the decision-making remains external to the agent script reduces readability, encapsulation, and transparency of the agent programs, and makes the control more opaque to the designer. For these reasons,

the selection functions hardcoded should be kept as simple as possible.

Default implementations are based on selecting the first available option, which is indeed a good example of simplicity. If we apply the default implementation also on this example, the first applicable unifier for `!go_order(L,M)` is `{Loc/italian, Meal/M}`, and the first applicable unifier for `!order(S,M,W)` is `{M/meal(veg,fish,white)}`.

3.3.3 CP-Nets and CP-Theories

In order to specify preferences, we start from the definition of CP-Nets given in [29]. Given a set of variables $X \in V$, each having a finite set of values x, conditional preference statements are in the form $u : x \succ x'$, where x, x' are assignments of a variable $X \in V$, and u is an assignment to a set of variables $U \subseteq V$ (parents of X). The interpretation of this statement is that given u, then x is preferred to x' all else equal, meaning, for all assignment s of the set of variables S, where $S = V - (U \cup \{X\})$, sux is preferred to sux', where sux and sux' are two *outcomes* (complete assignment) to all variables of V. CP-Theories are introduced in [166] to extend CP-Nets with *stronger conditional statements*. These include preferential statements in the form $u : x \succ x'[W]$, where $W \subseteq V$ which interprets that for all assignments w, w' to variables of W and assignments t to variables of $T = V - (U \cup X \cup W)$, then the outcome $tuxw$ is preferred to the outcome $tux'w'$. This means that given u and any t, then x is preferred to x' *regardless* of assignments to W.

Assuming Λ is a set of acyclic (with respect to parent-child relations) preference relations over variables of V, and considering o, o' are outcomes of V, then we say $\Lambda \models o \succ o'$ iff $o \succ o'$ satisfies every preference statement in Λ. Then o and o' can have one of the possible relations according to Λ: either $\Lambda \models o \succ o'$; or $\Lambda \models o' \succ o$; or $\Lambda \not\models o \succ o'$ and $\Lambda \not\models o \succ o'$. The third case means there is not enough information to prove either outcome is preferred.

Based on these definitions, two distinct ways for comparing outcomes are proposed in [29]:

- Dominance queries: Asking if $\Lambda \models o \succ o'$ holds, which is referred to as o is preferred to and *dominates* o'.
- Ordering queries: Asking if $\Lambda \not\models o' \succ o$ holds, which is referred to as o is preferred to o'.

Although ordering queries are weaker than dominance queries, they are still sufficient in many applications, and will be used in this work. In particular, if an outcome o is present such for all other outcomes o' we have $\Lambda \not\models o' \succ o$, then we say o is *undominated* or *most preferred* with respect to Λ.

All through this work, and for the sake of simplicity, only strict preferences $\succ$ are considered. Nevertheless, these semantics are shown to be extendable to weak preferences $\succeq$ and indifference $\sim$ in both CP-Nets and CP-Theories.

Embedding in BDI agents

To transform CP-Theories to a formalism that can be used with the Prolog-like inferential systems as those used in AgentSpeak(L) agents, one should look at what needs to be decided in the process of goal refinement. An agent may have dynamically interconnected beliefs about the environment, but when it is deciding on what is the most preferred approach to partially ground the variables of an event or goal in the form of e.g $!g(v_1, ..., v_n)$, only the parameters of that goal are relevant to the decision. Theoretically, in this approach we do not have only one CP-Theory, but each distinct goal/event has zero or more inferred CP-Theories from the set of all preference statements.

In this work, the preferences of an agent are presented in a different notation from that of CP-Nets and CP-Theories, but more similar to OCP-Theories in [66].

A conditional preference statement λ of the agent can be expressed in the form of inferential rules such as:

$$G \succ G' \leftarrow C$$

where G, G' are either belief predicates in the form $g(v_1, ..., v_n)$ and $g(v'_1, ..., v'_n)$, or triggering events in the form $!g(v_1, ..., v_n)$ and $!g(v'_1, ..., v'_n)$ (or any other type of trigger, $?, +, -$). Each v_i and v'_i can be either a (partially) ground term, a named variable or an anonymous variable (underscore, "_"), and C is an arbitrary logical expression that *activates* the preference statement if it can be proven to be true at the time of evaluation, which can include variables that appear on the left side of the $\leftarrow$. The set of all preferences of an agent is referred to as Λ.

With this definition, for each predicate G with the form $g(v_1, ..., v_n)$, we can denote its set of variables (or features or attributes) as $V_G = \{v_1, ...v_n\}$. To express a preference statement $u : x \succ x'[W]$ in this form, on a predicate G, assuming $G_X \in V_G$ is the variable of G corresponding to X, the set $G_U \subseteq V_G$ is the set of all variables corresponding to U, the set $G_W \subseteq V_G$ is the set of all variables corresponding to W and $G_T = V_G - (G_U \cup \{G_X\} \cup G_W)$, the statement can be presented as $G \succ G' \leftarrow true$, such that G_X is written as x, G'_X is written as x', all the variables of G_U and G'_U are written as their corresponding value in u, all variables of G_W and G'_W are written as anonymous variables (underscore) and all the variables of G_T and G'_T are replaced with named variables that have the same name in both G and G'.

Showing that these two forms of statements are equal is intuitive. For instance, given the statement of $g(x, u, T, _) \succ g(x', u, T, _) \leftarrow true$ and two (partially) grounded terms $g(t_1, ..., t_4)$ and $g(t'_1, ..., t'_4)$, we can infer $g(t_1, ..., t_4) \succ g(t'_1, ..., t'_4)$ iff we have $t_1 = x$, $t'_1 = x'$ and $t_2 = t'_2 = u$ and $t_3 = t'_3$ regardless of the values of t_4 and t'_4. By using induction we can see that the same can be inferred for any number or parameters that correspond to G_U, G_W, G_T or with any other rearrangement of the parameters.

Pure CP-Theory (and by extension CP-Net) statements can be expressed in the form of $G \succ G' \leftarrow C$ where $C = true$, but, as BDI agents are designed to

act in dynamic environments with incomplete information and uncertainty, and more often than not, a BDI agent has to react to changes in the environment and failures; simply using static CP-Theory statements is not sufficient for a BDI agent. This is addressed by the *activation* condition C of preference statements. This condition can be any arbitrary Prolog-like expression over the belief base of the agent. A preference statement is *active* if the context condition can be proven from the belief base of the agent. Intuitively this means that the left side of the rule holds true if the right side can be proven. This can drastically increase the expressivity of the preference statements in dynamic environments.

Definition 3.4 (Active Preference Statement)
At any point in the life-cycle of agent with a belief base B and set of preference statements Λ, a preference statement $G \succ G' \leftarrow C$ is active iff C is a logical consequence of B.

The next example further explores these type of preferences, and is an extended version of what is presented in the original CP-net paper [29] to facilitate comparison.

Example 3

Let us consider some preferences over the actions of our food ordering agent, starting from some preferences over the meal: (R1) for the main course, meat is preferred to fish if at an Italian restaurant; (R2) fish is preferred to meat if at a French restaurant; (R3) if the main course is meat, then a fish soup is preferred to vegetable soup; (R4) if the main course is fish, then a vegetable soup is preferred; (R5) for drinks, red wine is preferred to *any* other type of drink if vegetable soup is in the meal; likewise, (R6) white wine is preferred to *any* other drinks if fish soup is in the meal.

While multiple preferences are defined over the meal, still they do not translate directly to any of the events that the agent can handle. To fix this, we add a simple but important (meta-)preference: (R7) in the event of ordering *anything*, ordering something more preferred is also preferred to ordering something less preferred.We define at this point a few preferences about the restaurant: (R8) if the agent is already located at a restaurant, it is preferred to order at that restaurant (i.e. not to move) compared to any other restaurant, regardless of the meal; (R9) the combination of Italian restaurant with a meat main dish is preferred to any other restaurant and main dish combination if the agent is not already at another restaurant. The specification of these preferences can be seen in listing 7. A more detailed and practical explanation of how these statements are written as Prolog rules can be found in section 3.4.

We can draw as in Figure 3.2 the preferential relation graph associated to the predicate `meal/3` with respect to preferences in listing 7 and the beliefs in listing 6. The graph clearly suggests that, in this context, `meal(veg,fish,red)` is the most

```
1  (R1) meal(S,meat,W) >> meal(S,fish,W) :- at(italian).
2  (R2) meal(S,fish,W) >> meal(S,meat,W) :- at(french).
3  (R3) meal(fish,meat,W) >> meal(veg,meat,W) :- true.
4  (R4) meal(veg,fish,W) >> meal(fish,fish,W) :- true.
5  (R5) meal(veg,M,red) >> meal(veg,M,_) :- true.
6  (R6) meal(fish,M,white) >> meal(fish,M,_) :- true.
7  (R7) !order(M1) >> !order(M2) :- M1 >> M2.
8  (R8) !go_order(L,_) >> !go_order(_,_) :- at(L).
9  (R9) !go_order(italian, meal(S,meat,W))
10          >> !go_eat(L,meal(S,_,W)) :-  not at(L).
```

Listing 7: Preferences of Food-ordering agent

preferred (undominated) option. Note however that this graph would have been different if the agent had the belief `at(italian)` instead of `at(french)`.

Generalization

The preference statements in listing 7 have been chosen because they offer a good representation of the possible uses of the proposed method, and can be easily generalized. First of all, we observe that only R3 and R4 are pure CP-Theory preferences. The statements R1, R2 are conditioned on beliefs that are external with respect to the variables of the preference itself (in this case, the location of the agent). This type of conditional statements result in multiple preferential relations that may also be contradictory. For instance, if both conditions `at(french)` and `at(italian)` were true at any time, then the two preferences would be contradictory and the preference relations concerning `meal/3` would be unsatisfiable.

The statements R5 and R6 specify preferences that, although simple, can not be expressed in standard CP-Theories, as they give conditional preference to a value of a variable (drinks) over any other value of that variable (*erga omnes* preference); this can be very useful in cases where the domain of values of a variable are unknown at design time, but the designer is aware of a few values that are always either desired or to be avoided.

As it was already observed, R7 is a meta-preference. The condition of this statement does not consult the belief-base of the agent, but rather the preferences of the agent. It works as a mapping of a preference expressed as object level to a preference expressed as action level. In general, any preference over objects is implicitly referring to a certain domain of activities, and it is as such only a more compact representation. Preferences as R7 are needed to make explicit this connection. The technical aspects of this step will be explained more in detail in section 3.4.

The statement R8 is conditional over a belief that the agent may have (`at(L)`), and also connects the variable of this condition `L` (that will be grounded at run-time) to the preference relation itself, creating an interesting *parameterized*

preference structure capturing in this case "*I prefer the place I am already at*".

Finally, also R9 presents a statement that can not be expressed in CP-Theories, in which two different variables are part of the preference. This can be useful tool but also can easily lead to cyclic preferences, thus, this type of preference statement should be used with care.

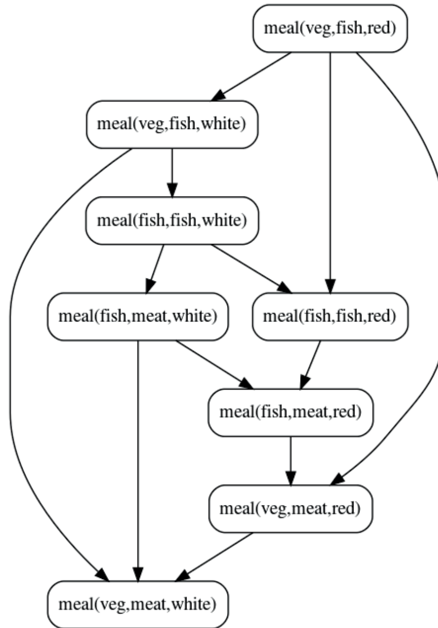

Figure 3.2: Preference Structure of the Agent over `meal/3`

3.3.4 Goal Refinement via Preferences

The integration of preferences into the BDI reasoning cycle can be implemented as a *unifier ordering* step prior the *plan selection*, that only applies when the triggering event contains unbounded variables. To achieve this, when the agent selects a partially unbounded event ϵ, first it needs to find all the relevant unifiers by consulting the plan library, and then find all the applicable unifiers by consulting the belief base. Afterwards, the agent can create a partial ordering between the applicable unifiers.

To extend the definition of section 3.3.3 to unifiers, assuming ϵ is a partially unbound event and $(\sigma\delta), (\sigma\delta)'$ are two applicable unifiers for ϵ, we say $(\sigma\delta) \succ (\sigma\delta)'$ satisfies a preference statement $G \succ G' \leftarrow C$ if $\epsilon(\sigma\delta)$ can be unified with G and $\epsilon(\sigma\delta)'$ can be unified with G'. Given Λ, a set of acyclic preference statements in this

form, then we say $\Lambda \models (\sigma\delta) \succ (\sigma\delta)'$ iff $(\sigma\delta) \succ (\sigma\delta)'$ satisfies every active preference statement in Λ. We can then say $(\sigma\delta)$ is *preferred* to $(\sigma\delta)'$ iff $\Lambda \not\models (\sigma\delta)' \succ (\sigma\delta)$. An then we can define the *undominated* or *most preferred* unifier:

Definition 3.5 (Most preferred unifier)
A unifier $(\sigma\delta)$ is referred to as an undominated or most preferred unifier for an event ϵ iff $\sigma\delta$ is an applicable unifier for ϵ and for every other applicable unifier $(\sigma\delta)'$ we have $\Lambda \not\models (\sigma\delta)' \succ (\sigma\delta)$.

Surprisingly, with this definition, finding the most preferred unifier is simple in a Prolog program as it matches well with how Prolog engines work. Normally in a Prolog program it is not easy to query if a fact holds with respect to every rule concerning that fact, but if a query about a fact fails, this means that all the rules about that fact have failed i.e, that the fact can not be proven. Thus, to find if a unifer $(\sigma\delta)$ is the most preferred one for an event ϵ, it is enough to ask if for every other unifier $(\sigma\delta)'$, the query $(\sigma\delta)' \succ (\sigma\delta)$ fails. Intuitively, running this query for every unifier of ϵ will result in finding the most preferred unifier. More details on the implementation of this algorithm is presented in section 3.4.

For simplicity, it is assumed that the agent uses the plan selection function typical in BDI frameworks, i.e. selecting the first applicable unifier for each event. This assumption means that the agent always uses an *undominated* or *most preferred* unifier to ground the variables of a (partially) abstract goal if this unifier exists, or reverts to the default behavior of selecting the first applicable unifier in case of inconsistencies with preferences that may result in situations that no unifier is the most preferred.

Example 4

Consider again the agent script given in example 1, with the beliefs of example 2, and with the preferences of example 3. Assume that this agent receives an abstract event `!go_order(L,M)`. Then, as in example 2, two applicable unifiers will be created. P1 will be applicable with the unifier `{Loc/italian, Meal/M}` and P2 with the unifier `{Loc/french, Meal/M}`. Because the agent has the belief `at(french)`, we can see that the relation:

```
!go_order(italian,M) >> !go_order(french,M)
```

can not be proven from the preferences (R8 and R9 can not conclude it) but the relation:

```
!go_order(french,M) >> !go_order(italian,M)
```

can be proven to be true (based on R8), so the `{Loc/french, Meal/M}` is the single most preferred unifier and P2 will be selected as this is the only plan applicable with this unifier. Next, when the sub-goal `!order(Meal)` is being considered, there

are 8 applicable unifiers for plan P3, assuming the `at(french)` belief still stands, and based on the given preference rules, the ordering in Fig. 3.2 will apply, and then the most preferred unifier will be `{S/veg, M/fish, W/red}`. This means that the abstract goal will be refined to `!order(meal(veg,fish,red))` and consequently plan selection will instantiate the plan associated to it (P3).

Example 5

To show how partially abstract goals would be grounded with this method, consider the same agent as before with the same set of preferences and beliefs, except this time the agent has the belief `at(home)` instead of `at(french)`, and the agent receives an event with ordering a meal with meat as the main course:

```
!go_order(L,meal(S,meat,W)).
```

This time plan P2 will be applicable with two unifiers:

```
{Loc/italian, Meal/meal(S,meat,W)}
{Loc/french, Meal/meal(S,meat,W)}
```

because the agent has the belief `at(home)`, the statement R8 is not active for any of the unifiers and based on the the statement R9, the Italian restaurant is preferred to any other restaurant as long as there is meat main course, so again while the relation:

```
!go_order(italian,meal(S,meat,W)) >> !go_order(french,meal(S,meat,W))
```

can be proven (based on R9), but the relation:

```
!go_order(french,meal(S,meat,W)) >> !go_order(italian,meal(S,meat,W))
```

can not be proven from the preference statements (R8 and R9 can not conclude it), so the first unifier will be the most preferred one and then P2 is selected with it. Next, assuming the `move_to` action works correctly, the belief `at(home)` will be retracted, `at(italian)` will be added to the belief-base, and the sub-goal `!order(meal(S,meat,W))` will be adopted. At this point (see the example in [29]), based on the preference statements and agent's beliefs, the most preferred unifier will be `{S/fish, M/meat, W/white}`, meaning the goal will be refined to `!order(meal(fish,meat,white))`, and then the plan for this goal (P3) will be instantiated by the plan selection.

An interesting point in this example is the interaction between preference statements R1 and R9. In normal CP-Nets, these two statements makes the network cyclic: the preference over main dish is dependent on the location (R1), and the preference over the location depends on the main dish (R9). But in this work such preferences can be defined for two reasons: (1) *framing*, meaning the two preferences, although being about the same variables, are defined in two different

frames of choice, and (2) *context*, meaning that as the agent resides and acts in a dynamic environment, it perceives changes and modifies its beliefs, which in turn modifies the agent's preferences, e.g. while the agent has the belief `at(home)`, R1 and R2 are not active and so they are not part of unifier selection process.

3.4 Implementation

This section presents a practical implementation of the transformation method from CP-Theory logic to Prolog logic proposed in section 3.3.3.

Preference operator

First, we need to express a preference statement $G1 \succ G2 \leftarrow C$, where $G1, G2$ are partially grounded terms with the same functor and arity. The mapping from CP-Theory statements to this statement is presented in section 3.3.3. This binary operator $\succ$ will be introduced both into the syntax of the agent programming language and as a binary predicate into the belief base of the agent. In the syntax, the operator $\succ$ will be denoted as `>>` making a relation such as $G1 \succ G2$ to be written as `G1 >> G2`. To write full contextually conditioned statements as $G1 \succ G2 \leftarrow C$, we can utilize the Prolog inference rules. The former preference statement can then be written as `G1 >> G2 :- C`.

Applicability

Next, as the method is implemented by utilizing the belief-base of the agent, a modification is needed to allow preference statements about different types of *goals* to be part of the belief base, as e.g. in the R8 statement from listing 7. To do this, a supplementary predicate *applicable*/2 is introduced. When a preference statement $G1 \succ G2 \leftarrow C$ is defined where $G1$ and $G2$ are triggering events e.g. achievement goal ($!G$), test goal ($?G$), the preference statement is transformed at compile/interpretation time to:

$$applicable(t, G1) \succ applicable(t, G2) \leftarrow C$$

where t is a predefined atom describing the type of the event, e.g: the preference statement R8 in listing 7 will be rewritten as:

```
applicable(achievement,go_order(L,_))
    >> applicable(achievement,go_order(_,_))
    :- at(L).
```

As this is a normal Prolog rule, it can be simply added to the belief base of the agent. Then, preference relations can be queried from any context, as e.g. the (meta-)preference R7 in listing 7, in which a preference relation is used as the condition of another preference, or, more importantly, to exploit Prolog queries to find the most preferred unifier for abstract events.

Optimality

Next, the algorithm should be implemented for finding an optimal outcome, that is, the most preferred unifier(s) for a partially abstract term. The algorithms originally introduced for CP-Nets and CP-Theories generate an optimal outcome by *sweeping* through the network from top to bottom (i.e., from ancestors to descendants) setting each variable to its most preferred value given the instantiation of its parents. While these algorithms are efficient and intuitive, they are not applicable for the transformed Prolog-like rules. Unlike CP-Nets and CP-Theories, the preferential structure of an agent is not static because of the presence of extra-contextual conditions; also, with the new form of statements, the parent-child relation of the variables is not explicit anymore. For these reasons, new algorithms are needed that do not rely on the hierarchy of the variables but instead utilize the backtracking feature of Prolog.

Looking at the definition 3.5, given a set of preference statements as Prolog rules in a belief base B, to prove that a unifier $(\sigma\delta)$ is the most preferred unifier for a partially unbound term (or event) ϵ, it is sufficient to prove that for every other unifier $(\sigma\delta)'$ of that term, the relation $\epsilon(\sigma\delta)' \succ \epsilon(\sigma\delta)$ is not a logical consequence of B. Intuitively with the semantics of Prolog, this means that this relation could not be concluded from any of the preference statements of B. With this, a simple algorithm that can find the most preferred (or undominated) unifier(s) for a partially unbound term is a backtracking search that goes through every possible (partial) grounding of that term to find one where there is no other (partial) grounding of that term which is more preferred to it. Such algorithm can be implemented by adding the Prolog rule presented in Listing 8 to the agent's belief base. The `copy_term/2` is a standard predicate in many Prolog implementations that unifies `G2` with a copy of `G` in which all variables are replaced by new variables.

```
1 most_preferred(G) :-
2     copy_term(G, G2), G,
3     forall((G2, (G2 >> G) -> fail; true)).
```

Listing 8: CP-Net reasoning algorithm implemented in Prolog

When this rule is queried with a partially unbound term `G`, first a copy of `G` is created as `G2`, then the term `G` itself is called, starting a backtracking search over all possible groundings of `G`, and then `forall/2` predicate starts a nested loop over all groundings of `G2` and fails if it can find a grounding of `G2` that `(G2 >> G)`, otherwise if no such `G2` is found it returns true meaning the current grounding of `G` is the most preferred one. Also with how prolog queries work, if there is more than one most preferred (undominated) grounding, asking for more answers will return them.

There is one final rule needed to make this algorithm work which should be added to the belief base of the agent: `T >> T :- !,fail.` which defines that a term can not be preferred to itself. With these rules added to the belief base, given a partially unbound term, we can run queries to find the most preferred unifier for that term. If we consider the example agent, querying the belief base with the term `most_preferred(meal(S,M,W))` will give the result in one answer with the unifier `{S/veg, M/fish, W/red}`.

Embedding in the decision-making cycle

Now that the belief base can answer queries about the most preferred unifier for a term, the next step is to allow the agent's reasoning engine to ask queries about the most preferred unifier for an event. To do this, the *applicable*/2 predicate that was defined before is used again. At compile/interpretation time, for each plan of the form $e : C \Rightarrow H$ a Prolog rule is added to the belief base of the agent in the form of:

$$applicable(t, G) \leftarrow C$$

where t is an atom that represents the type of the event e and G is the term associated with e. As an example, the rule for plan P1 is:

```
applicable(achievement,go_eat(L,M)) :-
    restaurant(L) & not at(L)
```

Intuitively, at any moment in run-time, by querying this predicate, we can retrieve all possible applicable groundings of an event that can be concluded from the plan library and the belief base. For instance, by querying the term `applicable(achievement,go_eat(L,M))` on the belief base of an agent with the beliefs, plans and preferences described in section 3.3, the agent obtains two answers: `{L/italian, M/M}` and `{L/french, M/M}`. Then, by using this predicate with combination of `most_preferred/2`, the agent can find the *most preferred applicable unifier* for an event. This is possible because the preference statements about events were already transformed with the `applicable` predicate. Considering our running example, by querying the term

```
most_preferred(applicable(achievement,go_eat(L,M)))
```

only one answer `{L/french, M/M}` will be returned. Now to embed the goal refinement step into the agent reasoning cycle. After the event selection step, if the selected event ϵ contains free variables, then the most preferred unifier(s) should be found for this event by querying the belief base of the agent with the aforementioned method, and the resulting answer(s) are sent to the plan selection function.

Complexity

The core of the method is the rule for the predicate `most_preferred/1`, and this rule has two nested backtracking loops over the possible groundings of the input term. In the worst case scenario, each two groundings of a term should be queried with all of the preference statements associated to that rule. Then, for a term T, if the there are n number of possible groundings at any time, and there are m preference statements over T, then, in the worst case scenario, the time complexity of finding the most preferred unifier will be $n^2 \times m$.

3.5 Discussion and Conclusion

This chapter contributes to recent efforts to integrate preferences into BDI agents. Despite the 'D' in the acronym, desires play a limited role in contemporary BDI agent platforms, as they are generally conflated to goals (procedural or declarative). This chapter showed that by interacting adequately with the belief base and plan library of the agent, abstract goals can be refined taking into considerations the agent's preferences. Stated differently, preferences act here as background desires modifying/impacting goals, playing the role in turn of contingent desires. (Note that in general the literature suggests that preferences are derived from desires [113]; for our purposes, however, we discovered that the two can be seen as filling the same functional niche.)

Although this work illustrated the use of preferences focusing on a single agent and on goal refinement, preference statements can be relevant in other contexts too. For instance, MAS frameworks allow agents to communicate and transmit their beliefs to each other. Leveraging the present proposal, because preference statements are implemented as beliefs, agents can directly communicate their preferences to other agents. This can be very useful e.g. in social simulation or social learning contexts, where the agents may need to decide to act (or not to act) depending on both their own and other agents' preferences. Another interesting use-case for this approach could be the implementation of normative agents, utilizing preference both to capture personal and societal norms (see the use of CP-nets for deontic logic in [112]).

Preference statements introduced in this work are in a form processable in Prolog logic programs. We have shown that a subset of this form can be used to express pure CP-Theory preferences. However, this new form can also be used to express contextually conditioned and parameterized preferences, resulting in much more flexibility than pure CP-Theories. Consider for instance the statement R8 in example 3: "*I prefer the place I am already at*"; that depends completely on the state of the agent in the environment and, if the environment is unknown and unpredictable, so will be the preference statement. Also, unlike CP-Theories that are fully qualitative, with the proposed form quantitative preferences can

be expressed by using arbitrary arithmetic equations in the context condition of preferences; e.g: consider a statement "*I prefer a cheaper restaurant to a more expensive one*" that can easily be expressed in this form with an arithmetic comparison as the condition of the statement. But this flexibility comes at the cost of static verifiability. Many of the preferences that can be expressed in the proposed representational model cannot be statically verified or predicted easily prior to run-time. The redeeming aspect of this problem is that static verification of dynamic agents is well-known to be a challenging task, especially in dynamic environments [79, 77]. This explains the existence of run-time verification approaches for agent programming frameworks, and probably the most notable works that adopt a (semi-)formal run-time model checking approach to verification are those of the AJPF/MCAPL framework [63, 62]. These approaches are in principle also usable for agents that are enhanced with preferences.

Finally, one of the main requirement behind this work is accessible usability. The transformation method from CP-Theory preference statements to Prolog-like programs has been conceived to enable its use almost directly with AgentSpeak(L)-like frameworks [26, 65, 126, 5]. Indeed, no extra reasoning component is introduced, all the preference reasoning and algorithms required for goal refinement is done through beliefs and inferential rules. Furthermore, unlike many of the works that embed preferences into BDI frameworks, e.g. [161, 162, 53, 122, 123], this approach does not require any extra annotation of the agent's script with information about effects of plans or actions and thus makes it more accessible for the designer.

A concrete Prolog implementation of ordering queries of CP-Nets/CP-Theories was presented, and a working proof of concept for this approach is publicly available as part of AgentScript[2]. Analyzing its complexity, we showed that the proposed algorithm run in polynomial time. Such worst-case scenario could be however reduced by optimizing the relative positions of preferential rules and groundings in the belief base, for instance by exploiting statistical information concerning their applicability or relevance for the decision-making cycle.

[2]https://github.com/mostafamohajeri/agentscript

Chapter 4

Interoperability and Automated Tests: DevOps

Previous chapters introduced different tools and approaches for modelling agents in norm-governed cyber-infrastructures, however, what typically holds such approaches back for real world applications is usability and adoptability for designers and modellers. Development solutions, such as testing and integration, undeniably play a central role in the daily practice of software engineering, and this explains why better and more efficient libraries and services are continuously made available to developers and designers. Could the MAS developers community similarly benefit from utilizing state-of-the-art testing approaches? The chapter investigates the possibility of bringing modern software testing and integration tools as those used in mainstream software engineering into multi-agent systems engineering. Our contribution explores and illustrates, by means of a concrete example, the possible interactions between the agent-based programming framework ASC2 (AgentScript Cross-Compiler) and various testing approaches (unit/agent testing, integration/system testing, continuous integration) and elaborate on how the design choices of ASC2 enable these interactions.[1]

4.1 Introduction

Software testing is attracting increased interest in industry [1] and it is one of the most used methods of software verification. One of the reasons of this success lies in the advancement and popularization in the software engineering community of methodologies commonly known as *DevOps*, in particular of techniques of automated testing in *continuous integration* (CI). Generally, CI refers to the facilitation provided by third-party tools for automating the build/test process of

[1] The material presented in this chapter refines and extends elements presented in [125].

a software. In recent years, online DevOps services such as TravisCI[2] and CircleCI[3] have been increasingly used by software engineers to improve the efficiency of their testing process, a practice which plausibly resulted in increased quality of the developed software.

Very recently, Fisher et al. [79] have suggested that testing approaches would be an important complement to formal approaches to MAS verification, if they could be automated and integrated in a seamless way into MAS development. In our view, seamless integration does not mean only that agent programmers are able to use the vast amount of software testing tools available to mainstream languages like Java or Python, but, more importantly, that they are also able to use (almost) language- and framework- agnostic online services as those used for CI. This chapter explores this idea, aiming to illustrate what the MAS community could gain by using industry standard testing tools and discussing what would be the theoretical and practical trade-offs for this choice. We investigate possible interactions of testing with agent-based programming, and its relation with other verification techniques. More concretely, we demonstrate various approaches to enhance the productivity of MAS development cycle in the ASC2 framework via mainstream software testing and integration tools, and elaborate on the design choices of ASC2 that affect the testability of agent-programs with the mentioned tools. Then, we explore on how this approach can be generalized for other MAS frameworks.

The motivation for this chapter arises from research conducted on data-sharing infrastructures (e.g. data marketplaces). At functional level, a data-sharing application corresponds to a coordination of several computational actors distributed over multi-domain networks. Those actors generally include certifiers, auditors, and other actors having monitoring and enforcement roles, ensuring some level of security and trustworthiness on data processing [174]. Typically distributed across several jurisdictions, networks may be subjected to distinct norms and policies, to be added to various infrastructural policies provided at domain level and *ad-hoc* policies set up by the users. Some of these norms, as for instance the GDPR, bind processing to conditions and specific purposes, but, more in general, all compliance checking on social systems requires to know and to infer (in case of a failure on expectations) *why* an actor is performing certain operations. Agent-based programming, and particularly the Belief-Desire-Intention (BDI) model, by looking at computational agents as *intentional agents*, provides the "purpose" level of abstraction available by design, and for this reason it is a natural technological candidate for this application domain.

The BDI model been extensively investigated as basis to represent computational agents that exhibit rational behaviour [96]. Recent works as e.g. [105] investigated various issues holding when mapping logic-oriented agent-based pro-

[2]`https://travis-ci.com/`
[3]`https://circleci.com/`

grams into an operational setting. In contrast, this chapter focuses instead on the *development practice* aspect: as soon as we attempted to program data-sharing applications as agents, we experienced the lack of mature software engineering toolboxes, thus hindering a continuous integration with the infrastructural-level components developed in parallel by our colleagues.

The document proceeds as follows: section 4.2 provides a background and related works on verification of MAS, in section 4.3 we introduce our approach on MAS testing in ASC2 framework with mainstream tools. An illustrative example of this approach is presented in section 4.4. Finally, section 4.5 provides the discussion and comments on possible extensions and future developments.

4.2 Verification of (Multi-)Agent Systems

Verification is a crucial phase in any software (and system) development process, and as such it has been addressed also by the Multi-Agent Systems (MAS) community. The survey presented in [8] provides an empirical review of over 230 works related to verification of MAS.

At higher level, approaches for the verification of autonomous systems fall into five categories [79]: (a) *model checking*, (b) *theorem proving*, (c) *static analysis*, (d) *run-time verification*, and (e) *(systematic) testing*. While the first four approaches (a-d) are considered formal or at least semi-formal, testing (e) is deemed to be an informal approach to verification. Further, MAS verification can be targeted at different levels, varying from fine-grained verification of agents at a logical level [11] to verification of emergent properties in a system [56]. Ferber et al [76] identifies three levels: (i) *Agent level* considers internal mechanisms and reasoning of an agent (ii) *Group level* consists in testing coordination mechanisms and interaction protocols of agents, and (iii) *Society level* checks for emergent properties or if certain rules and/or norms are complied within the society. In general, the choice of a verification method depends on the required level of verification, as e.g. formal methods may not be applicable for the verification of a large MAS with non-deterministic characteristics at the society level.

Most of the works on MAS verification point out that testing agent programs is far harder than testing normal software, on the grounds that agents tend to have more complex behaviors, and deal with highly dynamic and often non-deterministic environments (including other agents), on which they have only partial control [129]. A series of recent empirical results [168, 167] was used to conclude that, with respect to certain distinct test criteria, testing BDI agents can be practically infeasible. The *all-paths* criterion requires the test suite to cover all the paths of the agent's goal-plan graph; its application shows that the number of tests needed to run is intractable [168]. In subsequent work, the same authors study the minimal criterion of *all-edges*, requiring all edges of the goal-plan graph to be covered. While not *per se* infeasible, results show that even this criterion requires

a (too) high number of tests [167].

These observations can explain why much of the work in verification of autonomous systems and specifically of BDI agents have been towards the *formal verification* of agent programs, a mathematical process for proving that the system under verification matches the specification given in formal logic [24]. One of the most successful formal methods for verification of software agents has been *model checking* [41]. Model checking of BDI agents can be done as e.g. in [25] by translating a simplified version of AgentSpeak(L) to Java programs and using the Java Path Finder (JPF) verification tool. Probably the most notable works that adopt a (semi-)formal model checking approach are those of the AJPF/MCAPL framework [62, 77]; AJPF/MCAPL also relies on JPF to perform program model checking on agent programs developed in multiple JVM-based BDI frameworks by utilizing an implementation of the target language's interpreter. Nevertheless, although formal verification techniques as model-checking provide a high level of guarantee, they are typically both complex and slow to deploy [169].

A number of approaches to testing (that is, *informal verification*) have also been considered in the MAS literature. Some of those utilize model-based testing [137, 172] and rely on *design artifacts* such as Prometheus design diagrams [132] to generate tests and automate the testing process. Others consider a more fine-grained approach to verify intentional agents [73, 131], focusing on *white box* tests involving in the testing process the inner mechanisms of BDI agents (like plans and goals). This method of testing has however been criticized in [108] as being "too fine-grained", proposing instead to perform testing at a *module* level, that is, considering a set of goals, plans, and/or rules as a single unit. Still other works refer to *software testing* techniques applied on MAS development, focusing on testing agents and their interaction patterns as the main level of abstraction [45, 106]. At implementation level, such *unit testing* is performed in a *Jade* multi-agent system via the JUnit library. The distinct agent-roles that are present in the MAS are tested by means of *mock* agents that communicate with the implemented Jade agents to verify their behavior.

Levels of Testing

Software testing is generally categorized in four levels or activities: (a) *Unit testing* is done to verify different individual components of the software system in focus, (b) *Integration testing* verifies the combination of different components together, (c) *System testing* is done to test the system as a whole, and (d) *Acceptance testing* is done to check the compliance of the software with given end-users' and/or relevant stakeholders' requirements.

A categorization for MAS testing from a development-phase activity perspective has been proposed in [127], consisting of five levels: (i) *Unit testing* targets individual components of an agent, (ii) *Agent testing* aims at the combination of the components in an agent including capabilities like sensing its environment,

(iii) *Integration or Group testing* includes the communications protocols and the interactions of the agent with its environment or other agents, (iv) *System or Society testing* considers the expected emergent properties of the system as a whole (v) *Acceptance testing* for a MAS stays the same as their counterpart in software testing.

All these categorizations can be seen as guidelines to draw a conceptual line between what should be tested for what purpose and when, in the different phases of software development. This means that for each project it is up to the designer to decide e.g. what counts as units, what interactions are considered group and what are the properties of the system/society. Indeed, testing libraries like JUnit or online continuous integration services like TravisCI or CircleCI stay relatively agnostic on what type of tests are being done. We will follow here the same principle by allowing the designer to create each test suite with different scenarios containing one or multiple agents with varying types and allowing for flexible success/failure criteria.

Coverage

An important measure giving insights on the quality of a certain test suite in a given system is *coverage*. Software engineering proposes different criteria for coverage [128], varying from simple *line coverage* (denoting the percentage of the code that is covered by the test cases), to more sophisticated metrics like cyclomatic complexity [117], more commonly known as *branch coverage*. Intuitively, the more a program is covered by a test suite the more confident the designer can be about the behavior of the software. In fact it is a common approach to set a minimum coverage boundary for software projects and if coverage is below this limit the build chain is considered a failure even if the code compiles correctly.

Several works have studied criteria for testing in Agent-Oriented Software Engineering, and particularly in BDI-based agent programming [131]. However, the abstract mechanisms underlying any BDI-based reasoning cycle concerning e.g. treatment of plan context conditions, plan selection and failure handling, alongside the procedural specifications given in one agent's script (e.g. the agent's plans), result in complicated branching in the agent's effective code, a fact that makes defining what is actually covered by a test suite difficult [168, 167].

4.3 Approach

Instead of investigating dedicated tools for testing BDI agents, our motivation is to study under what conditions and how we can take advantage of existing software testing coverage tools, so as to enable an integration of BDI agent-based development with other types of development, occurring concurrently on a production-level system. This practical (and unavoidable) necessity motivated us

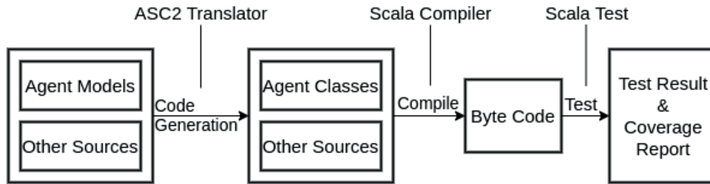

Figure 4.1: Compile/Test process of an ASC2 program with sbt

to overlook or put aside the warnings and issues indicated in the literature.

4.3.1 Testing Approach

In a typical *unit* or *integration test* of a computational entity under test (e.g. a class, a web service), the designer sets up an initial setting (e.g. one or multiple object instances, web services, a client), and then, based on certain invocations (e.g. function calls, access/service requests), a set of *assertions* are checked to verify the internal state, or some observable behavior of the tested entity, or its effect on the environment (e.g. function results, service responses, modifications of other entities).

Internal attributes (of objects or services) are generally harder to access and therefore to verify. Best practices of Test-Driven Development (TDD) address this issue by means of *Dependency Injection* (DI): the dependencies of each entity should be instantiated from outside the entity and then passed to it e.g. as parameters (typically to the class constructor in object-oriented programming). This allows the tester to isolate and observe the internal mechanisms of the entity under test by using "mocked" dependencies. To enhance testability, multiple components of ASC2 agents, including their belief base and communications layer, are injected as external dependencies.

In any certain situation, we can look at a single agent or multiple agents (a MAS) as a computational entity under test, and this entity has also a set of internal attributes, observable behavior, and possible interactions with its environment. The single agent or multiple agents under test can be instantiated from one or more scripts. The setting could include any other types of entities e.g. other possibly mocked agents, external objects, etc. The initial state of the agent(s) and of the other related entities defines the initial setting of the test, the invocation/probing action of a test suite is typically a series of messages sent to the agents. The expected effect(s), behavior(s) or state(s) of an entity rely heavily on the entity under test. For a small system including one or only a few agents, each message or the beliefs of the agent(s) may be needed to be verified, whereas in a complex system, the designer may only need to verify emergent pattern in the interactions of the agents or major shifts in the state of the system.

In our approach, we aim to allow the designer to utilize any off-the-shelf testing

```
1  +!init(W) : W > 1 =>
2      Nbr = "worker" +
3          ((#name.replaceAll("worker","").toInt mod W) + 1);
4      +neighbor(Nbr).
5
6  +!token(0) =>
7      #coms.achieve("master", done).
8
9  +!token(N) : neighbor(Nbr) =>
10     #coms.achieve(Nbr, token(N - 1)).
```

Listing 9: Token ring `worker` script in AgentScript DSL

tool (library, service, etc.) directly into their development chain, even more so to enable the designer to test their program via any standard build chain. In the case of the ASC2 framework, its current implementation is based on Scala, and we considered as target build tool *sbt*[4], which enables us to also use JVM/Scala testing libraries like *JUnit* or *ScalaTest*. We have then developed a sbt plugin[5] that —as part of the compile task—iterates over the scripts written in AgentScript DSL in the project sources and uses the AgentScript Translator to generate Scala implementations of the agents. Code generation is a standard part of build tools like sbt or maven, therefore, the generated sources are also managed by the build tool and are immediately available to rest of the project. The general overview of the *Compile/Test* cycle of an agent-based system developed via ASC2 and built by sbt is presented in figure 4.1. Note that this process is fully automated by sbt.

A MAS of this type can be started in two ways. After bootstrapping it as an empty instance of the MAS infrastructure, the designer can either use configuration files (e.g. JSON) to specify the agents of the system or alternatively, use lower-level code (e.g. Scala/Java) to manually spawn agents via their respective class in the generated code. In this work, we preferred the latter approach, as it provides better control over the test scenarios.

To complete our Compile/Test process, in addition to the *ScalaTest* library, we also used the *Akka Testing* library: at run-time, ASC2 agents are essentially Akka actor micro-systems and this library provides many convenient tools for testing actors. Both libraries are used out of the box and no modifications have been done to adapt them to the framework. With this configuration, each scenario to be verified can the written as a test suite in ScalaTest to test whether one or multiple agents behave as expected.

[4] `https://scala-sbt.org/`

[5] `https://github.com/mostafamohajeri/sbt-scriptcc`

4.4 Illustrative Example

To illustrate an application of our testing approach we consider a MAS constructed around a Token Ring system, commonly used in both distributed systems and MAS [126, 36]. This system consists of one master agent and W worker agents; at the start of the program the master sends an $init(W)$ message to all worker agents to inform them of the total number of the workers in the ring, each worker upon receiving this message finds its neighbor, forming a closed ring. Then, T tokens are distributed among the workers, each token has to be passed N times in the ring formed by workers. When all T tokens have been passed N times and this was reported to the master, the program ends.

4.4.1 Unit/Agent Testing

We will focus in particular on the script of the worker agents shown in listing 9. We perform the tests taking the standpoint of a *whitebox* test engineer, meaning that we test the script of the agent knowing its internal workings; nevertheless, the tests are still performed externally, we do not modify the script in order to test it[6].

Testing Successful Scenarios

By viewing the script in listing 9, we can see that the agent has a total of 3 plans for 2 separate goals. Theoretically, we need at least 3 tests to cover the successful execution of all the plans. However, while the success criteria for plans is simple (completion of execution), achievements of goals can be more complicated and the testing framework needs to provide the flexibility to define them. The success criteria for the `init(W)` and `token(N)` goals are quite different. In the latter the expected behaviour in both plans is an observable event, i.e. a certain `achieve` message sent by the agent to another specific agent. In the former case there is no observable behavior and the success criterion is a specific update of the agent's belief base.

The test specification we used for the worker agent can be seen in listing 10. In line 3 an empty MAS object is created. The criterion of success for `init(W)` plan depends on the agent's beliefs, therefore we need to be able to verify the internal state of agent's belief base. First we create an instance of `BeliefBase` class (line 4) and when the agent under test (`worker1`) is being instantiated (line 10), this object is injected in the agent as its belief base; with this approach at any point in the tests we can simply access the agent's beliefs to query them for verification purposes or even modify the agent's belief base for setting up test scenario states.

Only one agent (`worker1`) is under test and the other agents present in the suite can be mocked. As ASC2 agents are actor micro-systems, an agent can be

[6] `https://github.com/mostafamohajeri/agentscript-test`

```
class TokenRingWorkerSpec extends ... {

  val mas = new MAS()
  val verifiableBB  = new BeliefBase()
  val mockedMaster = testKit.createTestProbe[IMessage]()
  val mockedNeighbor = testKit.createTestProbe[IMessage]()
  val worker

  override def beforeAll(): Unit = {
    mas.registerAgent(new worker(bb = verifiableBB), name = "worker1")
    mas.registerAgent(mockedMaster, name = "master")
    mas.registerAgent(mockedNeighbor, name = "worker2")
    worker = mas.getAgent("worker1")
  }

  "A worker agent" should {
    "have its neighbor in its belief base after `!init(N)`" in {
      worker.event(achieve,"init(50)").send()
      mockedMaster.expect(GoalAchievedMessage())
      assert(verifiableBB.query("neighbor(worker2)") == true)
    }

    "send a `!done` to master on `!token(0)`" in {
      worker.event(achieve,"token(0)").send()
      mockedMaster.expect(event(achieve,"done").source(worker))
    }

    "send a `!token(N-1)` to its neighbor on `!token(N)`" in {
      worker.event(achieve,"token(10)").send()
      mockedNeighbor.expect(event(achieve,"token(9)").source(worker))
    }
  }
}
```

Listing 10: Test suite for the `worker` agent

mocked by a single actor. In lines 5 and 6, two *probe* actors are created to be the stand-ins for the master agent and (`worker1`)'s neighbor in the tests and they are then registered to the system (lines 11 and 12). This type of mocking gives us the ability to verify all the interactions that the agent under test may have had with these probe actors.

The rest of the test suite contains 3 tests, in the first test in line 18 a goal event `init(50)` is sent to the `worker1` agent and it is expected that after this goal is achieved (line 19), the belief base of the agent contains the belief defined

by the term `neighbor(worker2)` which is verified in line 20. In the next test, a goal message `token(0)` is sent to the agent (line 24) and then it is verified that the agent sends a `done` message to the master (line 25). The final test follows the same pattern by sending a goal message `token(10)` (line 30) and the verification includes a `token(10-1)` message to its neighbor (line 30). Note that in all the tests, the messages sent to the `worker1` agent do not specify any source, this is because in the script in listing 9, the source of the messages is not checked meaning it is not necessary to specify the source. As these tests are written in a standard testing library, build tools such as *sbt* can execute them in their build chain. By running the tests in the sbt shell we are able to see the output presented in listing 11 that indicates our program has passed this test.

```
[info] A worker agent should
[info] - have its neighbor in its belief base after `!init(N)`
[info] - send a `!done` to master on `!token(0)`
[info] - send a `!token(N-1)` to its neighbor on `!token(N)`
...
[info] All tests passed.
```

Listing 11: Output of the `worker` agent test suite

Testing Failure Scenarios

Successful executions are only a part of the full story. Indeed, in software testing it is acknowledged that covering *failures* is both more important and challenging, and thus requires more critical thinking by the test engineer [128]. Interestingly, failure tests are especially important in agent-based programming because failing under certain conditions may sometimes be the correct behavior for an agent.

Two failure tests are presented in listing 12. The first test sends a `init(W)` goal message to the agent with `W=-1` (line 3) but the first plan is applicable only for `W > 1` and the expected behavior of the agent in this situation is a failure which is verified by expecting a `NoApplicablePlan` message. In the second test, a goal message `unknown` is sent to the agent (line 8) for which the agent does not have any plans and it should reply with a `NoRelevantPlan` (line 9). Note that failure of a goal is not only reflected by the absence of an applicable plan or more generally failure in execution of a plan; similar to the success scenarios, the designer can define any other arbitrary criteria for a failure scenario.

Although we acknowledge that testing an agent program for every possible failure can easily become an infeasible task [168, 167], certain failures may be particular important for the designer to test, therefore there is value in enabling this possibility.

```
"A worker agent" should {
  "send a `NoApplicablePlan()` on `!init(-1)`" in {
    worker.event(achieve,"init(-1)").source(mockedMaster).send()
    mockedMaster.expect(NoApplicablePlan())
  }

  "send a `NoRelevantPlan()` on `!unknown`" in {
    worker.event(achieve,"unknown").source(mockedMaster).send()
    mockedMaster.expect(NoRelevantPlan())
  }
}
```

Listing 12: Failure tests for `worker` agent

4.4.2 Coverage

We explore at this point whether and how off-the-shelf coverage tools such as *scoverage*[7] can be used for code coverage analysis of agent programs written in ASC2, considering both statement and branch coverage aspect. To perform this we simply add the scoverage plugin to our project and generate a coverage report.

The coverage report produced for the worker agent by means of the previous tests is presented in Table 4.1. The `worker.Agent` row shows the coverage for the internal mechanisms of the agent, like e.g. *event handling*, while the other rows show the coverage report for each separate event, as an example, the `worker.token_1` refers to an event `token` in `worker` agent with 1 parameter. The branch coverage report mainly concerns conditional statements in the generated Scala code of the agent and should be regarded only as informal information about the coverage of the main script.

These results show that our tests indeed covered most of the behaviors that the agent might have. In fact, by exploring the coverage analysis we can see the reason for which the `worker.token_1` has less coverage: the missed branch can be explained by the fact that the tests did not include any scenario in which the `token(N)` plan fails. Also note that while the example script did not contain any sub-goals or conditional statements in the plans, ASC2 Translator generates sub-goal adoptions as function calls and translates conditional statements to their counterpart in the underlying language, therefore, coverage tools like *scoverage* are able to calculate the correct number of covered and total possible branches for deeper goal-plan trees.

[7] `http://scoverage.org/`

Component	Statement Coverage %	Branch Coverage (Covered/Total)
worker.Agent	93.5	6/6
worker.init_1	93.5	2/2
worker.token_1	80.2	3/4

Table 4.1: Coverage analysis of the worker agent

4.4.3 Integration/System Testing

Even following the guidelines on categorizing different levels of testing in MAS [127], there is no definite technical distinction in place. Typically test libraries provide mechanisms such as annotations for the designer to label test suites with its (their) related level(s) to orchestrate their execution. As illustration, we consider an integration test to verify a token ring MAS system consisting of the previously mentioned worker agents and a master agent. The test suite is reported in listing 13.

```
1  class TokenRingIntegrationSpec extends ... {
2
3    //a communication layer that records a trace of the interactions
4    object recordedComs extends AgentCommunicationsLayer { ... }
5
6    val token_pattern = "token\\([0-9]+\\)".r
7    val done_pattern = "done".r
8
9    "A token ring MAS with W = 100, T = 50 and N = 4" should {
10     "have 250 `token(X)` and 50 `done` message" in {
11       // create the agents
12       mas.registerAgent(new worker(coms = recordedComs),
13         num = 100)
14       mas.registerAgent(new master(coms = recordedComs),
15         name = "master")
16       // invoke the system
17       mas.getAgent("master").event(achieve, "start(50,4)").send()
18       // verify the interactions
19       watchdog.expectTerminated( mas, 10.seconds )
20       assert(recordedComs.trace.count(token_pattern.matches) == 250)
21       assert(recordedComs.trace.count(done_pattern.matches) == 50)
22     }
23   }
24 }
```

Listing 13: Integration test suite for the token ring system

The test will be centered around the interactions between agents and the state of the system in a specific setting of our token ring. The token ring is defined with 100 `worker` agents and 1 `master` agent (lines 12-13), and, to be able to verify the exhibited interactions, we use dependency injection to initialize all the agents by means of an overridden instance of the communication layer (line 4), created to record every message passed in the system into a list.

To invoke the system, a `start(T,N)` is sent to the `master` agent (line 15). We are interacting with the `master` from a *black box* perspective: although the event `start(T,N)` is exposed, the internal mechanisms of this agent are assumed to be unknown.

Three criteria are verified for this system. Firstly, we consider a system level performance based criteria as we expect the system to be terminated under 10 seconds (line 17). Next, we use two known expectations from a token ring system to verify the correct execution of the system: at the end of execution, there should be (a) T number of `done` messages and (b) $T \times (N + 1)$ number of `token(X)` messages in the trace. The interaction verification statements are presented respectively in lines 18-19. Recalling the flexible definitions of testing levels, note that these integration/system test could be considered from the perspective of `master` agent as a unit/agent level test possibly with mocking the `worker` agents. Similar to previous tests, running this suite via sbt yields the output in listing 14.

```
[info] A token ring MAS with W = 100, T = 50 and N = 4 should
[info] - have 250 `token(X)` and 50 `done` message
...
[info] All tests passed.
```

Listing 14: Output of the token ring integration test suite

4.4.4 Continuous Integration

The proposed approach for testing can be easily combined with online CI services. This process generally includes utilizing source repositories like Github[8], CI services like TravisCI and code analysis services like Coveralls[9]. The only step needed to set the CI cycle for an ASC2 project is to configure the source repository of the project in a way that the automated CI cycle is triggered on every `push` to the repository. This can be done by adding a configuration file that provides information for the CI service how to compile and test the project via sbt.

Following this method, a MAS project does not need to be only located in a single source repository. For instance, different types of agents can be developed in different projects by separate teams and only be used as dependencies in the development of the system. We believe this is an interesting practical innovation, improving the scalability of MAS projects with respect to their development.

[8] `https://github.com/`
[9] `https://coveralls.io/`

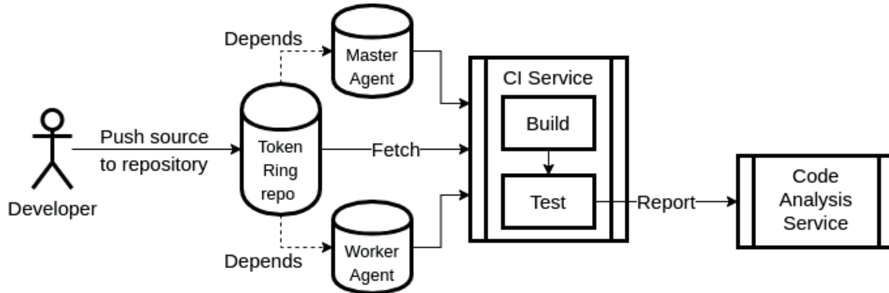

Figure 4.2: Continuous integration applied on a Token ring program whose master and worker agent scripts are located on other repositories.

An overview of an example CI process for the token ring is presented in Figure 4.2 in which the sources of `worker` and `master` agents are located in separate repositories, and a third token ring repository uses them as dependencies. When the system designer pushes the project to the repository, the CI service fetches the source and compiles and tests it via sbt and records the results[10]. Then, the code coverage report is committed to the code analysis service[11].

4.5 Discussion and Conclusion

Despite the critical points/observations concerning MAS testing raised in the literature, in this chapter we provide several support arguments for using mainstream testing tools for MAS and agent-based programming, by means of a concrete use case. We implemented a multi-agent system reproducing a token ring benchmark with the framework ASC2, and then we run tests (success, failure, coverage) at unit/agent level as well as at integration/system level.

At the unit and agent level (unit testing) we performed tests concerning events, plans and goals. The somehow unexpected result of the experiment is that such an approach does not neglect the theoretical complexity of BDI agents but it truly offers a complementary tool for their development. We were able to test successful (plan) completions, internal states and the belief base, failures, and fine-grained interactions. These possibilities can be seen as offering constructs mapping e.g. to declarative and procedural goals in BDI agents [170]: the designer can define the achievement/failure of a goal not only in terms of completion/exception of a plan, but also as determined by any arbitrary indicator internal or external to the agent. This showed that testability of agent programs defined in a framework is closely related to the design choices of that framework.

[10] `https://travis-ci.com/github/mostafamohajeri/agentscript-test`
[11] `https://coveralls.io/github/mostafamohajeri/agentscript-test`

At the integration/group and system/society level (integration testing) we performed tests with simple verification criteria, but these criteria can easily be extended to more sophisticated and realistic interaction analysis and verification methods developed by the MAS community [27]. Additionally, we illustrated how the proposed approach enables the MAS designer to take advantage of continuous integration (CI) services without extra effort. This is particularly important for MAS designers that require to integrate and test their work continuously with other projects.

There is an additional benefit of using mainstream test tools for BDI agents, and especially for frameworks that are based on higher-level logic-based DSLs. Those frameworks generally map primitive actions to constructs specified in a lower-level programming language like Java. By using a testing process compatible with both higher level models and lower level implementations, the testing process can be more efficient and seamless for the designer specially if the agent models are only a part of a project that includes other computational entities that are being developed alongside the agents.

An issue in using mainstream test libraries for a BDI framework with a logic-based DSL is the disparity between the high-level agent DSL and the lower-level language used for the tests. This can be addressed by either developing approaches to write tests in the high-level DSL or creating interfaces for the low-level language to enable the test engineer to implement tests at a proper level of abstraction. In this work we have taken the latter approach. The intuition behind this choice was that frameworks based on cross-compila-tion [65, 141] produce source codes that can be directly integrated within standard build tools.

Can our results be generalized to other agent programming frameworks? Motivated by the success of works like AJPF/MCAPL [62] that provides model checking for multiple BDI frameworks, as a future study we intend to explore how to apply this approach to a wider range of MAS frameworks. Yet, we can already trace some higher-level considerations. The answer, at the unit/agent level, depends on compilation and the execution model of those frameworks. For frameworks like Jade and JS-son [105], that use mainstream programming languages to define agents, these tools should be compatible out of the box with minor effort [106]. For cross-compilation-based frameworks like Astra [65] and ASC2 [126] it is only the matter of tooling (e.g. build tool plugins) to allow them to use mainstream testing tools. For interpreter-based frameworks like Jason [26] and GOAL [99], because they require their own dedicated reasoning engines and execution environment, testing via such tools may prove to need more work and possibly modifications to the framework. This issue may be not so problematic, as there are already many works that propose dedicated testing and debugging approaches for interpreter-based frameworks [108].

At the integration and system level, and also with respect to compatibility with CI services, generally externalized to the execution of the tested entity, we believe it is possible to consolidate other frameworks regardless of their compile/interpret

model. This could lead to seamless integration testing of systems defined in each framework with mainstream software testing tools or dedicated ones.

In perspective, our overarching research concerns socio-technical and complex multi-domain infrastructures; we believe that Agent-Oriented Software Engineering can be a powerful technical tool with robust theoretical foundations for designing, modelling, implementing and testing such systems. Enhancing their development cycle goes with a seamless integration of multi-agent systems into modern infrastructures. This is a critical requirement to utilize the full potential of MAS in a real production-level setting.

Chapter 5

Introducing Norms to Agents: Normative Advisors

This chapter explores the next step in modelling norm-governed systems by introducing a modular architecture for integrating norms in autonomous agents and multi-agent systems. As the interactions between norms and agents can be complex, this architecture utilizes multiple programmable components to model concepts such as adoption of multiple personal and/or collective, possibly conflicting norms; interpretation and qualification between social and normative contexts; possibility of intentionally (non-)compliant behaviors; and resolution of conflicts between norms and desires (or other norms). The architecture revolves around *normative advisors* that act as the bridge between intentional agents and the institutional reality. As a technical contribution, a running implementation of the architecture is presented based on the ASC2 framework and eFLINT norm reasoner.[1]

5.1 Introduction

Designing software agents that can reason with norms—technical instances of *normative agents*—requires evidently having a suitable computational model for reasoning with norms. This is a challenging task because norms are more than a set of formal rules extracted from a legislative text: they emerge from multiple sources with different degrees of priority, they require interpretation to be encoded and qualification to be applied within a social context. Furthermore, they continuously adapt, in both expression and application [18].

Previous proposals to embed norms in multi-agent-systems (MAS) have focused on extending the agent architecture (usually based on beliefs-desires-intentions, or BDI) to allow for forms of normative reasoning [68, 33, 156, 49]. Two general approaches can be identified in the literature. One can encode norms as facts, and

[1]The material presented in this chapter refines and extends elements presented in [134].

axiomatize normative reasoning with dedicated rules, to leverage the inferential engine provided with the agents; in this case, no modification needs to be applied to normative agents compared to non-normative agents. In contrast, in frameworks like BOID [33] or B-DOING [68], explicit normative components like obligation (O) are considered to play a role in the decision-making cycle, requiring a modification of said cycle. In both cases, the agent is considered as a unity from an execution point of view: the agent script is a program, usually interpreted, and its internal reasoning cycle executed by a single thread. This work starts from the observation that this constraint is a rather strong one, not needed, and not always the most suited. Rather than an event-based reactive architecture based on a single event-queue and scheduling, individual agents may be each implemented as a system of concurrent actors, provided with some form of organization (e.g. event dispatching) and interaction mechanism (e.g. messaging).

Normative agents offer a context in which this way of design gains more viability, for several reasons. On *content* level, multiple normative sources may be concurrently relevant, and/or multiple interpretations of the same normative sources may be available (e.g. retrieved from previous cases), and these may be possibly conflicting. Enabling to maintain those in a modular fashion is a suitable precondition for update/adaptation actions, where norms can be changed on the fly, and agents may decide at run-time e.g. to change the relative priority between normative components. On *method* level, there is still an ongoing debate on what is the most adequate representation model for norms, and on proper methods for normative reasoning (e.g. managing conflicts). Enabling the recourse to external tools, and supporting programmability of the coordination level, greatly empowers modelers/programmers/designers to test and compare different choices. Finally, at *functional* level, most of the knowledge instilled in norms concerns a full social system; only a part is contingently relevant to the agent. Designing the system so that it distributes the inferential load at best (and at need) externally from the decision-making is seemingly the most efficient option.

Contribution For these reasons, this work proposes an abstract architecture that encapsulate norms (namely encoded in terms of normative relationships of Hohfeld's framework) in a MAS. The architecture centers around *normative advisors* that can be utilized by (other) agents in the MAS as a sort of council about the institutional state of affairs and normative relations between agents, highlighting the mapping between the social and institutional views of the environment. Agents may resort to personal or to collective advisors, depending on the decentralization constraints set up by the designer. As a technical contribution, we present a practical implementation of this architecture that relies on the AgentScript BDI framework (ASC2) [126] for programming agents, and norm specification framework eFLINT [158] for encoding norms.

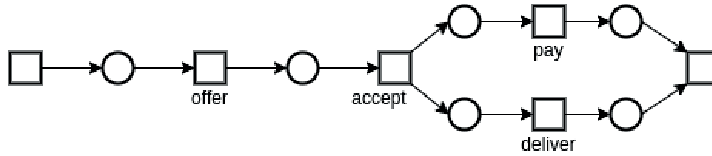

Figure 5.1: Sale transaction as a *Petri net* workflow

Related works The B-DOING framework [68] explores logical relations between belief, desire, obligation, intention, norms and goals in agents and their interactions like conflicts and possible appraoches to balance them in agent's behavior. Similarly, the BOID architecture [33, 133] proposes a belief, obligation, intention and desire architecture with a feedback loop to take the effects of actions before committing to them. These studies (and many others, e.g., [60, 156]) propose extensions to the BDI architecture to add (regulative) norms as part of the agents' mind and to resolve conflicts with pre-defined rules as part of the agent's reasoning cycle.

The normative BDI architecture presented in [49] proposes different contexts for an agent: mental, functional and normative contexts, plus "bridge rules" between them. Their approach aims at creating maximally compliant agents and focus on solving conflicts between norms at the time of adoption by using pre-defined conflict resolution rules. The work presented in [35] propose an approach for ethical reasoning in MAS by programming *ethical governors*. Their method is close to ours in the sense that they also take advantage of extra agents in a MAS, but they focus on machine ethics used in the decision-making of one agent.

Structure of the document Section 5.2 gives background on the core components that the proposal uses, providing some detail on the AgentScript/ASC2 and eFLINT frameworks. Section 5.3 lays out the theoretical framework for the proposed architecture, whereas Section 5.4 describes details of its implementation. Section 5.5 reflects on the capabilities of our implementation, suggests future directions, and draws connections with related work.

5.2 Core components

To illustrate our approach, we will consider as a running example a marketplace environment consisting of buyer and seller agents. This target domain can be seen as an abstract model of many real-world domains, e.g. data market-places and more in general data-sharing infrastructures, electronic trading infrastructures, etc. The process model of a individual sale transaction—prototypical example of bilateral contract—can be represented as a workflow through a Petri-net, presented in Figure 5.1. A seller offers a buyer an item for a certain price. If the buyer

accepts the offer, then the seller is expected to deliver the aforementioned item to the buyer, and the buyer is expected to pay the seller the agreed upon price (in any order). The workflow is a simplified representation of the normative mechanisms in place during an actual sale transaction. Furthermore, it does not consider the intentional aspects on the agents during the transaction, e.g. based on which desires or goals the agents may be willing to engage in the transaction, as these concepts stay external to norms.

5.2.1 Intentional agents

The intentional agents defined in this chapter intuitively are implemented in ASC2. Continuing with the example, Listing 15, presents the script of a buyer agent. The initial beliefs (lines 1-3), initial goals (line 5), and plan rules (line 7 and onwards) are the components of the script. The script is further explained in Section 5.4.3.

5.2.2 Norms and Normative (Multi-Agent) Systems

Following Gibbs, we consider norms as "a collective evaluation of behavior in terms of what it ought to be; a collective expectation as to what behavior will be; and/or particular reactions to behavior, including attempts to apply sanctions or otherwise induce a particular kind of conduct" [85]. This definition is relevant to our purposes as it gives primacy to action (rather than to situations, as in most deontic logic accounts). In the context of multi-agent systems, and even more of in MAS, an action-centered approach is intuitively more suitable, as actions are the only means agents have to intervene on the environment, and by which determine normative consequences.

Categories of Norms Norms are traditionally distinguished between *regulative* and *constitutive* norms [144, 21, 149]. Regulative norms regulate behaviors that exist independent of the norms and are generally prescribed in terms of *permissions*, *obligations* and, *prohibitions* (e.g. traffic regulations). Constitutive norms determine that some entity (e.g. an object, a situation, a certain agent, a certain behaviour) "counts as" something else, creating a new institutional entity that does not exist independently of these norms. (for example, the concept of marriage or money as a legal means of payment). The concept of institutional power is particularly relevant in the context of constitutive norms, as it is used to ascribe institutional meaning to performances (e.g. raising a hand counts as a bid during an auction). A conceptual framework that instead contains both deontic and potestative dimensions is the one proposed by Hohfeld [100], whereas deontic logics, although much more popular, by definition focuses on regulative norms.

```
needed_item("Book1").
fair_price("Book1", 5).
have_money(10).

!init(#sale_advisor.getClass, "sale.eflint", "BuyerAdvisor").

+!init(AgentType,EFFile,Name) =>
    #spawn_advisor(AgentType, EFFile, Name).

+offer(I ,P) =>
    #achieve("BuyerAdvisor", perform(offer(Source, Self, I, P)));
    !consider_buying(Source, I ,P).

+!consider_buying(Seller, I, P) :
  needed_item(I) && fair_price(I, FP) &&
  P =< FP && have_money(M) && M >= P =>
    #tell(Seller, accept(I, P));
    +pending(accept(I, P)).

+acknowledge(accept(I, P)) : pending(accept(I, P)) =>
    -pending(accept(I, P));
    #achieve("BuyerAdvisor", perform(accept(Self, Buyer, I, P))).

+duty_to_deliver(Seller,Buyer,I) :
    Source == "BuyerAdvisor" && Buyer == Self =>
        +expected_delivery(Seller,I).

+delivery(Sender, I) : expected_delivery(Sender, I) =>
    -expected_delivery(Sender, I);
    #achieve("BuyerAdvisor", perform(deliver(Sender, Self, I)).

+duty_to_pay(Buyer, Seller, P) :
    Source == "BuyerAdvisor" && Buyer == Self  =>
        !pay(Seller, P).

+!pay(Seller, P) : have_money(M) && M >= P =>
    #pay(Seller, P);
    #achieve("BuyerAdvisor", perform(pay(Self, Seller, P)).

+!pay(Seller, P) => ... ALTERNATE APPROACH TO PAYMENT ...
```

Listing 15: Buyer agent script as ASC2 program

Normative systems

The term normative system can be applied to a system of norms, and a multi-agent system guided by norms. Our work builds upon the second standpoint, and more precisely, we apply the so-called *normchange* definition of normative MAS system by Boella et al. [22]: “a multi-agent system together with normative systems in which agents on the one hand can decide whether to follow the explicitly represented norms, and on the other the normative systems specify how and in which extent the agents can modify the norms”. This definition does not assume any particular inner workings of the agents except that they should be able to somehow *decide* whether to follow the norms or not and they should be able to *modify* them. Furthermore, there is no assumption about the representation of the norms, except that they should be *explicit* (i.e. a ‘strong’ interpretation of the norms [20]) and *modifiable*.

The eFLINT norm language

The eFLINT language is a DSL designed to support the specification of (interpretations of) norms from a variety of sources (laws, regulations, contracts, system-level policies such as access control policies, etc.) [158, 157]. The language is based on normative relations proposed by Hohfeld [100]. The type declarations introduce types of *facts*, *acts*, *duties* and *events*, which together define a transition system in which states—knowledge bases of facts—transition according to the effects of the specified actions and events. The transitions may output violations if an action triggered a transition with unfulfilled preconditions (e.g. only sellers can make offers) or if any duties are violated in the resulting state[2].

Listing 16 shows an eFLINT specification for our running example. The `Actor` and `Recipient` clauses and `Holder` and `Claimant` clauses of act- and duty-type definitions establish constructs mapping to Hohfeldian power-liability and duty-claim relationships. The `Creates` and `Terminates` clauses describe the effects of actions when performed, enabling reasoning over dynamically unfolding scenarios. An instance of `offer` can be performed without any pre-conditions and it holds when there is a `seller` instance. The act `accept` is only available after an offer: accepting a non-existing offer is considered a violation of the power to accept offers. Acceptance of an offer creates the two act instances `pay` and `deliver` which can be performed in any order. The duties express that the `pay` and `deliver` actions are expected to be performed by their respective holder after they are created as part of the `accept` action. As described in Listing 16, no violation conditions are associated with the duties.

[2] In eFLINT, a permitted act may result in changes to normative positions and that the ability to trigger these changes constitutes a power; therefore act frames can be seen as specifying at the same time powers and permissions, and, absence of powers map to prohibitions (as e.g., in access control methods). Other computational frameworks based on Hohfeld propose a more explicit separation between deontic and potestative categories [150].

```
1  // fact definitions
2  Fact buyer
3  Fact seller
4  Fact item
5  Fact price Identified by Int
6
7  // act-type definitions
8  Act offer Actor seller Recipient buyer
9    Related to item, price
10   Holds when seller
11   Creates
12     accept(buyer, seller, item, price)
13
14 Act accept Actor buyer Recipient seller
15   Related to item, price
16   Creates
17     pay(buyer, seller, price),
18     duty_to_pay(buyer, seller, price),
19     deliver(seller, buyer, item),
20     duty_to_deliver(seller, buyer, item)
21
22 Act pay Actor buyer Recipient seller
23   Related to price
24   Terminates
25     duty_to_pay(buyer, seller, price)
26
27 Act deliver Actor seller Recipient buyer
28   Related to item
29   Terminates
30     duty_to_deliver(seller, buyer, item)
31
32 // duty-type definitions
33 Duty duty_to_pay
34   Holder buyer
35   Claimant seller
36   Related to price
37
38 Duty duty_to_deliver
39   Holder seller
40   Claimant buyer
41   Related to item
```

Listing 16: eFLINT Specification for *Sale Transaction* norms

5.3 Normative MAS via Normative Advisors

Our approach is based on the introduction of *normative advisors* that enable intentional agents to communicate with an external norm reasoner. We assume the parent agent is a BDI agent, i.e. it has the capabilities to reason with beliefs,

desires and intentions. The tasks of maintaining an institutional perspective (state) and reasoning about specific sets of norms is delegated to the advisors. The advisors are initialized with a particular norm specification and maintain an institutional perspective on the environment, which is continuously updated at run-time. A normative advisor is therefore viewed as maintaining (inferential mechanisms necessary to operationalize) a *norm instance*. Both regulative and constitutive norms are taken into account. The normative (institutional) state of the world is stored in a way that can both be queried and updated at any time. An update can generate normative events that the agent is to be notified about. Through the normative advisors, a social agent acquires various capabilities to interact with norms. As a consequence, norms interactions become programmable parts of the agent, realizing our goal of using norms for behavioural coordination between agents and for specifying qualification processes from social context to norms. With such an infrastructure, an agent becomes:

- able to *adopt* or *drop* any number of norm sources as norm instances;
- able to *qualify* observations about their environment as normatively relevant updates, and conversely to *respond* to normative events by acting accordingly in their environment;
- able to *query*, *update*, *revert* and *reset* a normative state of any norm instance;
- able to *receive and process* or *ignore* normative events (e.g. new claims and liberties)
- able to *follow* or *violate* normative conclusions (e.g. obligations) or query responses (e.g. permissions and prohibitions)
- able to *modify* any of the above abilities at run-time.

Normative reasoning occurs based on these inputs—triggered by queries or updates— with all conclusions made available as internal events to the advisor. Note that an agent can have multiple advisors for different (instances of) sets of norms. An agent is free to qualify observations about events in the environment, other agents' actions, its own beliefs and actions—or any combinations of these—and report the resulting observations to the relevant normative advisors. In other words, this infrastructure makes possible a rich, recursive interaction between behavioral decision-making and normative reasoning. The proposed model supports a number of programmable concepts applicable to different functions:

1. *Perception*: which internal/external events are received and processed or otherwise ignored;

2. *Reaction and planning*: what are the relevant reactions to an event, which reactions are applicable in the current context and which reaction is the most preferred one to execute;
3. *Norm adoption*: when and how to adopt or drop a set of norms;
4. *Qualification of social context*: how an event or query is qualified, i.e. which is its normative counterpart for each norm instance;
5. *Querying*: when and how the normative state of an instance needs to be queried (e.g. for compliance checking);
6. *Reporting*: what events/updates are reported to which norm instances;
7. *State change*: how a normative event changes a norm instance's state;
8. *Event generation*: what normative events are created as the result of an instance's state update;
9. *Qualification of normative concepts*: which events should be raised as the result of what normative conclusions reported by a norm instance.

To concretize the proposed approach, we will discuss at higher-level why it is feasible to implement a system meeting these requirements by utilizing an AgentSpeak(L)-like BDI framework (AgentScript/ASC2, in particular) and a norm reasoner that can store an updatable and queryable normative state, generating events on updates (eFLINT, in particular). Perception, planning and execution are basic core functions of reactive BDI agents as those specified via AgentSpeak(L), i.e. when an event is received, the agent performs a series of actions as reaction. Qualification can be encoded as part of planning: what reaction is selected for an event (or a series of events) in any context signifies how that event is qualified. Norm adoption, querying and reporting intuitively become part of this reaction. Note however that querying can also be part of planning, as a query response may affect what reactions are applicable. State changes happen internally to the norm instance as the result of reporting, and then normative events are generated, which are in turn qualified as events by the agent, creating a full circle. Finally, if both the BDI framework and norm framework allow for run-time changes, as is the case with ASC2 and eFLINT, then all aspects are changeable and dynamic.

5.4 Implementation

This section describes an architecture for advisors and discusses how the ASC2 BDI framework and the eFLINT normative reasoning framework are used to implement the architecture. The eFLINT framework is used to implement the norm base. The advisors as well as the intentional agents that employ them are

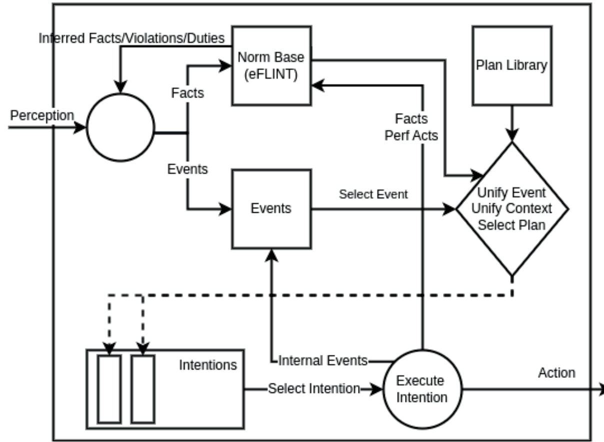

Figure 5.2: Normative advisor's architecture

defined in ASC2. Our implementation benefits from the modularity provided by ASC2, allowing easy replacement of different parts of the agent[3] and the Java API provided by eFLINT.

5.4.1 Normative advisor architecture and decision-making cycle

Figure 5.2 illustrates an overview of the architecture of a normative advisor. It is inspired by the BDI architecture of Jason [26] and resembles ASC2's default functional architecture in Figure 2.1. In effect, in this architecture a normative advisor can be seen as a BDI agent in which the (typically Prolog-like) belief-base is replaced the norm reasoner, thus, logical reasoning of the agent is replaced with normative reasoning. Apart from the differences between a logical reasoner (e.g. Prolog) and a norm reasoner (e.g. eFLINT), the main architectural differences of an advisor with a typical BDI agents are: (1) the belief-base (in this case, the norm-base) of the agent can generate more than just belief-update (or fact-update) events, it may now also raise duty events, act (enabled/disabled) events and violation events upon which the agent can react according to its plan library; (2) from the execution context of a plan alongside fact-update actions (`+fact` and `-fact`), there can now be act-perform actions (`#perform(act)`). These differences arise from the the fact that unlike a logical reasoner like Prolog that typically uses

[3] ASC2 uses Dependency Injection, meaning most of the dependencies of an agent (e.g. belief-base, communication layer) are passed to it; also, Inversion of Control, meaning higher-level objects (e.g. agent) define a generic control flow and call into lower-level potentially customized objects (e.g. belief-base) to concretize the execution.

backward-chaining to infer facts based on queries, the eFLINT framework also produces information in a forward-chaining manner, thus generating more events for the advisor to process. Despite these modifications, the core of the AgentScript DSL, and the capabilities of the framework, like goal adoption, communication, and performing arbitrary primitive actions, remain the same as with intentional agents.

Let us analyse a decision-making cycle of the advisor. When the advisor receives an external or internal event, if it is a fact-update, it will be sent to the norm base. If the event is an achievement or test event, it will be sent to the event queue. Events are taken from the event queue by an event-selection function, at which moment the head of the event is matched with the plan library to find all the relevant plans. The context conditions of relevant plans are checked against the normative state of the norm base in order to select only applicable plans. Then, a plan selection function selects one applicable plan and turns the plan into an intention, and, consequently, an intention selection function chooses intentions for execution. If the body of the plan includes any fact-update actions (`+fact` and `-fact`) or act performance (`#perform(act)`), then these are sent to the norm base. Whenever there is any update committed to the norm base, there could be multiple new events or new facts derived by the normative reasoner that are sent back to the advisor as internal events.

These new capabilities are also the result of replacing the Prolog reasoning engine with the eFLINT reasoner. Any Boolean expressions in the DSL can now refer to pre-defined predicates corresponding to eFLINT keywords for querying the norm base: `holds/1` is used to check if a fact (or act, duty, etc.) holds, `enabled/1` whether the preconditions of an act hold, and `violated/1` checks if a duty was violated. A comprehensive list of possible interactions with the eFLINT norm reasoner is given in the next subsection.

5.4.2 eFLINT norm base implementation

The eFLINT language is implemented in the form of a reference interpreter in Haskell[4]. As discussed in [158], the interpreter can run in a 'server mode' in which it listens to requests on a certain port and produces responses according to some API. A layer has been developed on top of the server to maintain multiple server instances as is need for supporting multiple advisors with a norm base each. An eFLINT server instance can receive the following **requests**:

- *Fact creation/termination/obfuscation.* A created fact (instances of fact-types, act-types, duty-types and event-types are referred to as facts) is set to 'true' in the knowledge base, a terminated fact to 'false' and any existing truth-assignment is removed when a fact is obfuscated.

[4]Publicly available online `https://gitlab.com/eflint/haskell-implementation`.

- *Triggering an action or event.* Instances of act-types and event-types can be *triggered*, resulting in the effects of the action or event manifesting on the knowledge base (`#perform` in Listing 17). These effects can be to create/terminate/obfuscate certain facts, as listed in the corresponding (post-condition) clauses of the type declaration of the triggered action/event. Note that, because of the synchronization mechanism, multiple actions/events can be triggered at once.

- *A query in the form of a Boolean expression.* The expression is evaluated in the context of the current knowledge base and can be used to establish whether a certain fact holds true in the current knowledge base, whether an action is enabled (`holds` in Listing 17) or whether a duty is violated, etc.

- *The submission of a new type declaration* or *the extension of an existing type*. Both have the effect of modifying the norms in the sense that the underlying transition system is modified.

Every request can be associated with additional context information in the form of truth-assignment to facts that override any conflicting assignments in the current knowledge base (e.g. the current UNIX time). This mechanism can also be used to provide truth-assigments for 'open types' (see below). An eFLINT instance generally operates *synchronously*, i.e. will only send out information in **responses** to requests[5], updating the sender upon the following:

- Any created, terminated, and/or obfuscated facts. Note that this includes changes to facts that are (or were previously) derived from other facts and in this sense were indirectly modified by the incoming request

- Any changes to normative positions regarding duties, i.e. whether a duty is no longer held by an actor or whether a duty is now held by an actor (e.g. `-duty` and `+duty` in Listing 17). Violated duties are also reported as such.

- Any changes to normative positions regarding powers, i.e. which actions became (or are no longer) enabled. If the incoming request was triggering one or more actions that were not enabled, the effects of the actions still manifest, but the violations are reported.

- In response to a query, the reasoner responds with the result of the query (state is unchanged).

- If the incoming request requires the evaluation of a fact for which no truth-assignment is given and which is an instance of an 'open type'—a type for which the closed world assumption does not hold—then an exception is

[5]If necessary, a clock event can be triggered periodically, possibly resulting in synchronous updates.

```
1 +?permitted(A) : enabled(A) => #respond(true).
2 +?permitted(A) => #respond(false).
3
4 +!perform(A) : enabled(A) => #perform(A).
5 +!perform(A) => #tell(Parent, failed(A)).
6
7 +duty(D) => #tell(Parent, D).
8 -duty(D) => #untell(Parent, D).
```

Listing 17: AgentScript specification of norm advisor.

raised and reported to the sender of the request. Evaluation is interrupted and the state remains unchanged.[6]

All changes to facts' truth-assignment, normative positions and violations register as internal events in the normative advisor (as shown by Listing 17), which will process and possibly report them according to its script.

5.4.3 Spawning and interacting with advisors

Scripts of normative advisors (written in ASC2 DSL) run on top of the advisor architecture and give the programmer access to the norm reasoner, both providing its input in the form of queries and updates and responding to the normative events the reasoner generates. In such sense, advisors functionally act as "bridges" or chain of transmission between institutional and social realms. Listing 17 shows a basic script for an advisor in our running example. The advisor has four test-goal plans related to acts and two related to duties. The query `+?permitted/1` receives an act and responds with `true` if the given act is "permitted" according to the underlying norm reasoner—in the case of eFLINT "enabled"—and `false` otherwise. The agent has similar plans to submit (or not) the performance of acts (`+!perform/1`) to the norm reasoner. The last two plans are triggered when the internal norm reasoner creates (`+duty/1`) or terminates (`-duty/1`) a duty. The advisor informs their parent of these changes. The fragment demonstrates that observations about created and terminated duties are communicated to the intentional actor (`Parent`) and that an action **A** can only be performed when it is enabled according to the norm reasoner (or fails otherwise); however this script does not demonstrate all the features possibly delivered by the architecture such as internal events for violations, enabled/disabled acts, and asserted/retracted facts.

To demonstrate spawning and interacting with a normative advisor, consider again Listing 15 in which a script for a buyer agent is given. Together, Listings 15,

[6] The exception can be used by the parent of the advisor to acquire the missing information, e.g. from another agent in the MAS.

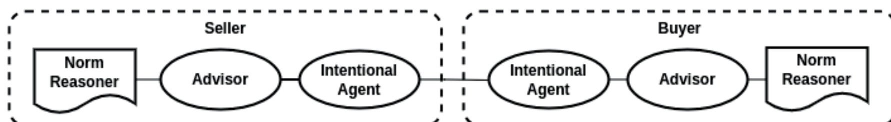

Figure 5.3: Market-place model

16, and 17 show the DSL code for buyer agent in the market-place as presented on the right side of the Figure 5.3. The buyer agent spawns a normative advisor, which in turn spawns an eFLINT server (norm reasoner). The buyer has its own beliefs and desires: there is a specific item that it needs (`needed_item/1`), it has a belief about the fair price (valuation) for that item (`fair_price/2`) and it has a belief about how much money it possesses (`have_money/1`). When this agent receives a `+offer/2` message about an item and its price, first it interprets it as an `offer` act and sends it to its advisor. Next, it adopts a goal of `consider_buying` that item for the price. This goal has one plan associated to it, which checks if the agent actually needs that item, if the price is considered a fair price and finally if the agent has enough money to buy that item. If this is all true, it sends a `accept/2` message to the agent that made the initial offer. Unlike before, this alone does not constitute performing the normative `accept` act. Instead, it waits until it receives a `+acknowledge/1` message from the seller before communicating acceptance to the advisor. This extra-institutional step for the buyer to qualify the act of `accept`, is an example of context-based qualifications in intentional agents.

When the `accept` act is submitted to the norm reasoner, the two previously mentioned duties of `duty_to_pay` and `duty_to_deliver` are generated and sent by the advisor to the intentional part of the Buyer. For the `duty_to_deliver` the agent is the *claimant* (it holds the expectation of performance); it could be that the agent asks the seller agent at this point to deliver the item, but instead, with the implicit assumption that the Seller agent is also compliant to the same set of norms, this agent simply adds this expectation to its belief-base and only when it has an observation of `delivery/2`, it will remove this expectation and send the `deliver` act to the advisor.

For the `duty_to_pay` the agent is the duty-holder (it has the obligation to perform) and reacts to this duty by adopting the goal `pay/2` (meaning it has the desire to be compliant). There are two plans for this goal, the first one is straightforward and is applicable if the agent has the required amount of money which then it will simply pay the Seller and submits this act to the advisor. However, the second plan (not implemented) is applicable if the agent does not have enough money, which means it needs to find alternative paths to relieve this duty, e.g. by returning the item, borrowing from another agent or even asking another agent to pay the seller instead. Specifying these alternatives requires extensions to either the agents, to the norms, or to both. Rather than working out

one or more of these alternatives, we consider different interesting opportunities of extending this straightforward example and reflect on the design of advisors in the next section.

5.5 Discussion

This paper presents an approach to embed (constitutive and regulative) norms into a MAS in a modular and versatile manner, enabling autonomous agents to reason with norms. In this section we discuss this approach, its connection to certain requirements for a normative MAS, while reflecting on the illustrated example.

Inline with MAS, and distributed computing in general, we consider **consistency as a consequence** of how a system is set up rather than it being ensured by the framework through which the system is built. This allows for a kind of partial consistency that enables freedom for local deviations that are not harmful to the overall system behavior. In our approach, norm adoption and qualification is done by each individual agent, such that their view on the normative state of the world is dependent on both their script and their (bounded) perception. Particularly desirable for social simulations, we can define agents that adopt and follow the same norms but have different conclusions on the normative state of affairs because they have had different observations. Alternatively, agents do not have to follow the same norms but might still be able to behave in a coordinated fashion. An example of the latter in our sales example is a buyer that believes, on top of the existing norms of our example, that deliveries should be done before payments. The buyer can behave according to their own norms without violating the norms adopted by sellers, even though the norms are different.

As presented in the previous sections, our running example shows how **coordination** between agents is achieved by adopting norms and deciding whether to comply with norms. The example relies on the agents wanting to comply, and therefore exhibiting coordinated behavior. In more adversarial environments, additional *enforcer* agents can be added to provide (positive and negative) incentives to comply. For example, our marketplace can be extended with an agent that acts like a market authority. By responding to violations raised by their advisor(s), the market authority can apply **ex-post enforcement** of norms on the market's participants. For example, a buyer refusing to pay can receive a warning or, in the case of continued non-compliance, be banned from the market altogether. This discussion further demonstrates the versatility of our approach: it does not impose *a priori* centralized/decentralized governance or ex-ante/ex-post enforcement. Instead, our approach gives the system designer the flexibility to choose, design and test what their system requires.

Referring to the requirements in Section 5.3, the notion of **adopting** was illustrated in the simplest form with the buyer agent in Listing 15 with the

`#spawn_advisor/3` to adopt a norm as an initial goal. The agents also have the `#despawn/1` action to and **drop** an advisor. However, by adding extra mechanisms in the agent's script, more complex archetypes can be modelled, e.g. the agent may be programmed to keep a score for a certain norm's (and advisor's) utility to decide if it is an effective norm to keep adopted.

The notion of **qualification**—necessary to fill the gap between computational forms of law and software [18]— can performed at various stages, thanks to the multitude of programmable layers in our approach. An example of qualification in the sale transaction is how a seller agent perceives a *pay* act from a buyer agent. While represented as an act in the norms, in the social reality many different actions can be perceived as a payment, e.g. both direct cash payment or indirect 3rd-party transaction (bank transaction) can be qualified by sellers (and authorities) as the act of paying. While direct cash payment is simpler to qualify for the agent, a bank transfer can be more complicated. This qualification rule could have been encoded in the script of an agent. For example, a bank agent can update a seller that they have received new funds as part of a completed transaction. The seller can then determine whether these funds constitute a payment by a buyer for a particular item, and inform the corresponding advisor. The same qualification can also be performed purely within norms, specifically as in eFLINT, actions and events are synchronized such that preconditions and effects of transitions are effectively 'inherited'. In this way, explicit 'counts as' relations between performed actions (transitions) can be specified. Listing 18 shows for instance how a transaction event in a banking system is connected with (qualified as) a payment action in our running example. This means the intentional agent only needs to indicate to the advisor that the original event `transaction_completed` was triggered which will automatically be inferred as performance of a `pay` act.

```
1 Fact account
2 Placeholder sender    For account
3 Placeholder receiver  For account
4 Event transaction-completed
5   Related to sender, receiver, price
6   Syncs with pay(sender, receiver, price)
7     When buyer(sender) && seller(receiver)
```

Listing 18: eFLINT fragment connecting a bank transaction to the `pay` action.

The notions of **query**, **update**, **revert** and **reset** are already afforded by the norm reasoner where query and update are typically provided by most norm frameworks. However, eFLINT can be used to reason about the compliance of historical, hypothetical, and – most relevant here – dynamically developing scenarios: it relies upon a declarative component that lays out the norms in the

form of a labelled transition system and an imperative component that describes traces in this system. This means that similar to belief queries and revision, the agent is able to query and revise (assert/retract) institutional facts. But, unlike physical state, institutional state is revertible as for example, an agent may notice that its observation about performance of an act was not correct or even it wants to infer hypothetical effects of performance of an act before reverting them.

Another important legal/normative requirement is **adaptability** to new (interpretations of) norms. In our approach, such adaptation can be achieved in at multiple ways. Firstly, apart from spawning new (and despawning old) advisors to start using the new interpretation or encoding of a set of norms, ASC2 agents are able to modify their script at run-time to change the interactions between institutional and social reality, this is true for both intentional agents and advisors. An agent can keep an advisor and its institutional state but instruct the advisor to change how certain events should be handled by modifying its plans. This type of modifications are also present in other BDI frameworks such as Jason. Secondly, an existing advisor can be instructed to update the norm source of an instance it has spawned by adding new type declarations or extending existing types. For example, a violation condition can be dynamically added to the payment duty by submitting the fragment `Extend Duty duty_to_pay Violated when <EXPR>` for some Boolean expression, like a parameterized timeout event. These types of modifications are particularly interesting as a future work to explore a principled approach for studying changes in the norms such as issues about consistency between variations of norms and impact of norm changes in social simulations.

However, this does not represent the whole adaptation problem as modification of rules can be much more complex. From the prospective of jurisprudence, laws can be separated into primary and secondary rules [92]; primary rules are about how individuals should act (or not act) and duties that should be fulfilled. Secondary rules are about how primary rules should be created and modified. Typically, normative frameworks (including eFLINT) mainly focus on primary rules while rarely taking secondary rules into account. This means that rather than claiming to solve the whole adaptation/modification problem, we provide a potential starting point to tackle this challenge and believe the fundamental architectural design is adequate to implement other more complex approaches to norm adaptation.

The notions of **receive and process/ignore** and **follow/violate** for normative conclusions connect directly to the concept of autonomy in the agent. All of these are are already afforded by ASC2 on the language level (or AgentSpeak(L) in a broader sense) as receive and process/ignore, and, follow/violate are simply a matter of implementing the plans in the agent's script that define the reactions to such conclusions. Then, as the intentional agents' language and execution cycle are not modified in this architecture, intuitively, autonomy of the agents is also not demoted by integration of norms, particularly in comparison with any BDI agent that does not integrate norms. However, as a future work, these

concepts– specially follow/violate– should be encoded in a more expressive and transparent manner. This can be done, for example, by utilizing declarative constructs such as preferences on the language level (see [124]) to have an explicit, yet programmable way of ordering between intentional (e.g. desires, goals) and normative (e.g. obligations) dimensions of the agent.

5.6 Conclusion

This work presented a theoretical framework for embedding norms in a MAS. It is generally acknowledged that agents in a MAS vastly benefit from utilizing norms for more effective/efficient coordination. Here it was further argued that norms, embodied as institutional views of the state of the environment, need normative advisors to facilitate the bridging between institutional and extra-institutional realms. The proposed architecture included using a BDI framework and a norm reasoning framework for creating normative advisors and was shown to address the main requirements of normative (multi-agent) systems as identified by the community. A practical running implementation of this architecture[7] using mostly off-the-shelf tools was presented via a market example to further illustrate the applicability of the approach.

As autonomous agents, norms, and their interactions deal with notions and constructs hard to concretize and on which it may be hard to reach an agreement, they may have different definitions and usages in different scientific communities. Alongside the proposal of the architecture and tools in themselves, this work assumed a high priority for flexibility as a requirement in frameworks utilized in designing normative (multi-agent) systems by proposing multiple programmable components varying from pure context-free and abstract norm specifications to perception/action layer of intentional agents. These components aimed at satisfying the higher level requirements of normative agents and (multi-agent) systems without putting any constraint on the language or logic used in components.

[7] publicly available:
`https://github.com/mostafamohajeri/asc2-data-marketplace-example`

Chapter 6

Example 1: Coordination in MAS via Norms

Until this point, this thesis has introduced tools and methodologies for modelling and designing norm-governed (multi-agent) systems. This chapter utilizes these tools to illustrate, implement and analyze a model of a Data Market-Place as a normative multi-agent system. Norms in this context are used as a fully distributed coordination and monitoring mechanism between participants of the market. The goal of the chapter is two-fold: (1) illustrating how such models can be valuable assets for system designers and policy makers by providing insights and generating design artifacts, and, (2) as part of the DL4LD project, proposing a flexible yet powerful schema and method for development of a data market place.

6.1 Introduction

The intrinsic and potential value of data in many different aspects of human society can not be overestimated, from financial [93] to scientific research [61] and healthcare [146], every important sector is impacted, and arguably potentially improved in their effectiveness by utilizing data-oriented approaches. Intuitively, the importance of data-sharing between parties is also rising, which in turns creates concern about data security, privacy, legal, monopoly and many other issues, especially in contexts in which parties may not have mutual trust, or may be even competitors [44]. This context creates the need for governance approaches that go beyond single organization scenarios.

The idea of Data Market Places (DMPs) is one that addresses these issues. A DMP is a membership organization in which each member can only perform actions (such as transferring data and execution of algorithms) based on previously constructed contractual agreements [171, 145], consortium policies and legislative regulations (e.g., GDPR), and for specific purposes. Intuitively, such a framework can induce collaboration and add much needed trust between parties, ensuring

that their artifacts (data, algorithm, processing power, infrastructure, etc.) will be utilized only in the intended manner.

Although there are practical works on implementing such system, like for instance the Mahiru framework [160],low-cost andefficient modelling of a DMP remains an important (yet overlooked in the literature) challenge to facilitate the design of a DMP. This chapter utilizes the tools and approaches proposed in the previous chapters to model a DMP as a Multi-Agent System. Starting from agreements between participants, the DMP can create services to enforce control and governance over the flow and utilization of artifacts between actors, and guiding them through their roles in the consortium, defined by the application contract, all while ensuring compliance to legislative regulations.

The model of DMPs in this chapter is based on the collective works[1] presented in [171, 145] and earlier ideas of a multi-domain data-sharing infrastructure in [86] and the Service Provider Group [87]. Furthermore, there have been previous works that have utilized Agent-Based approaches to model DMPs [60, 174] with a focus on specifying the model unlike this chapter which mostly is concerned with the modelling framework.

6.2 Background

To model a complex system as a MAS we must further study the literature on the connection norms such as contracts and regulations and multi-agent systems. This connection has been studied extensively in the normative MAS (nMAS) [22]. In a complex system with heterogeneous agents that may have conflicting goals, norms can provide an efficient way of coordination between agents typically in the form of agent organizations [114]. Most of the studies in nMAS are concerned with the theoretical aspects like the interactions between governing norms and individual agents' desires and goals; many of which were covered in Chapter 5.

However, there are also works that introduce practical approaches for embedding some degree of control in a MAS by enforcement of norms. An overview of these approaches can be found in [50], analyzing norm enforcement architectures based in criteria like supporting automatic enforcement, different levels of norms that describe actions and/or state of affairs, and dynamicity of the norms, in the sense that they can be adopted, dropped or changed. They also introduce practical criteria like execution efficiency, and centralized and decentralized nature of enforcement.

The authors of [50] also introduce the MaNEA, a norm-enforcing architecture aimed at controlling norms in open MAS. The MaNEA architecture utilizes a distributed model of agent organizations that includes norm manager nodes with prescriptive norms in the form of deontic directives (obligation, permission, and prohibitions), and norm enforcer nodes that can observe the system, detect

[1] Conducted in the same research group as this thesis.

violations, and respond to them with punishment/reward methods. Their approach differs from ASC2 and its normative extensions firstly because they focus on controlling a MAS as a whole, they assume full information of every action and message within the system from the perspective of enforcers nodes, while in the ASC2's architecture —and arguably in real life– this is not the case. Secondly, they only take into account prescriptive norms in form of deontic directives, while this thesis' approach goes beyond that by also utilizing norms as a coordination mechanism between agents, using norm representation and reasoning methods as given external components (and possibly changeable with others).

The authors of [9] introduce an approach called accountability-driven organization programming technique (ADOPT) to improve accountability within a MAS; their approach focuses more on the coordination and collaboration aspects of norms. In a setting in which multiple agents have to collaborate and perform specific sub-goals to achieve a higher-level goal, they propose the use of organizations of which each agents firstly has to agree to be a part of, and secondly it has to agree to specific sub-goals that it will need to perform. This work is close to ASC2 in that they also assume local nodes in a MAS that rely to agents for what they are ought to do, but where they only assume the purpose of collaboration, this thesis also takes into account the prohibitions that each agent may have, i.e., what they are ought not to do.

6.2.1 Compliance Management Framework

Another related research domain is the one addressed by Compliance Management Frameworks (CMF). In any organization, there is always the concern to verify if the business activities of the organization are compliant with governing rules and policies. This typically includes encoding the relevant rules into some normative specification and utilizing a reasoner for automated verification of normative specifications over business process specifications. There are multiple CMFs introduced in the literature. In [94, 95] multiple CMF frameworks are studied and a set of evaluation criteria are introduced for CMFs categorizing the frameworks. The criteria include: (1) the process life-cycle phase (design, run, and auditing time) a framework focuses on, and the orientation of the framework (focusing on formal verification or business oriented); (2) the expressiveness of the normative specification language, focusing which of the normative constructs the language supports; (3) how the framework creates the link between business models and normative specifications; and (4) the level of support a framework with respect to modeling, linking, compliance checking, and handling violations.

An example of CMF is the COMPAS framework presented in [74] that is designed for service-oriented architecture-based systems. In COMPAS, Linear Temporal Logic (LTL) is used to specify compliance requirements of a system and Business Process Management and Notation (BPMN) to identify the business process model. Then, formal model-checkers are used for design, run, and auditing

time verification of the compliance specifications over business process specifications. Like CMFs, the proposed approach of this thesis is also concerned with the compliance of a business process (albeit implicit in this work) with relevant norms. What makes COMPAS –and most other CMFs – different from ASC2's architecture is that the focus in a CMF is on one organization and its business model, however, by modelling the system as MAS, the focus is shifted towards possible interactions that may happen between multiple different actors with possibly conflicting goals in the presence of multiple normative sources.

6.3 The Model of DMPs

A DMP is a membership organization in which members are able to share (or buy/sell) different artifacts such as data, algorithms, and computation power. The concrete collaboration model i.e., the flow of artifacts and execution of computations within the DMP is based on predefined contractual agreement between all members. Given an agreement of collaboration in the form of an *application*, the DMP infrastructure should provide services and guidance to all participants and support them in playing their role in compliance with the agreement while making sure other participants are also compliant.

Apart from the contractual agreements, a DMP should also make sure that the activities of all parties are in compliance to general market place rules and constitutional regulations such as those about privacy (e.g., GDPR). However, this can also be seen as if the market place itself, and the governing regulatory bodies (e.g., governments, competent authorities) are already involved in every interaction and their conditions are (implicitly or explicitly) part of every agreement. A high-level view of a DMP can be seen in Figure 6.1.

Data sharing applications are at the core of operationalization of a DMP. An application defines the flow of processes between collaborating participants, from an organizational standpoint, this translates to defining the duties each participant has to fulfill and restrictions they have to take into account for the application to be performed successfully and in a compliant manner. Once an application is agreed upon by participating actors, there are multiple ways to orchestrate and schedule required actions of the actors. The simplest approach is to create a centralized setting, where there is one center that monitors the actions of the actors and notifies them the next action they have to perform. Although the simplicity of a centralized approach is desirable, it depends on very strong assumptions. Firstly, there is the assumption of observability of all actions where it may not be the case in a real setting. Monitoring a distributed system comes with a cost, and this cost is higher if there is one central node that is monitoring all of the system. As opposed to this, distributed and autonomous monitoring, e.g., actors monitoring their local ad-hoc connections is a much more feasible and scalable approach. Secondly, it assumes the presence of a fully concretized plan (a series

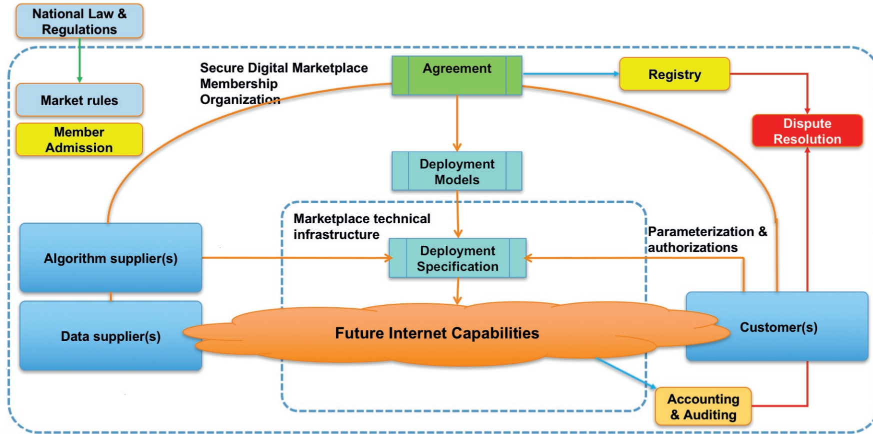

Figure 6.1: A high-level view of a Data Market Place

of actions), where in a real setting it may not be the case. Creating fully concrete plans in a setting with many actions and actors is expensive. Furthermore, actors may need to be autonomous with respect to *how* they will perform their part, e.g., delegating parts of tasks or creating local plans based on higher level partial plans in case of failures or to just lower costs while still acting compliant to overall policies. Furthermore, with central planning, the planner node needs to take the all the internal policies of all participants into account to be effective, but in reality the actors may require their policies to remain private resulting planning to be much less feasible.

To solve these issues, the DMP architecture mode proposed in this chapter utilizes a fully distributed control mechanism, where the centralized market actor only defines high-level conditional and context-based duties for actors and provides them with high-level specifications of conditional powers or abilities that they have to perform those duties. Still, the issue of monitoring and accountability remains, having each actor only aware of what it should do will negatively effect trust and cooperation between actors. As it was mentioned in Chapter 5, apart from letting actors know what they should (and not) do, norms also define expectations that actors can have from others. For a full distributed monitoring setting, the proposed architecture utilizes an ad-hoc monitoring approach, where each actor has expectations, and monitors only the actions in the system that it is also a part of. In summary, between two actors A and B, if A has a duty to send a stream of data to B, then B is the only actor that is aware of this duty and has an expectation of receiving that data stream. Hence, in effect, it is B that is monitoring A and holding it accountable. This chapter argues that distributed governance and control, plus an effective logging and auditing mechanism that

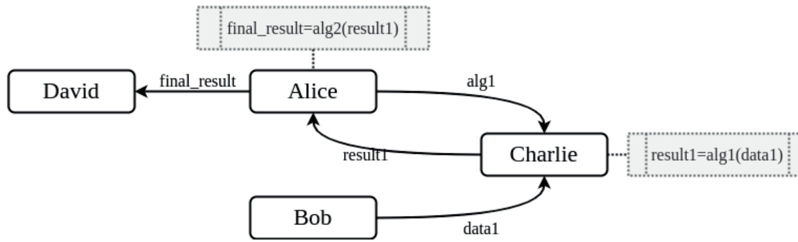

Figure 6.2: DMP Example Scenario

can lead to ex-post enforcement, can be much more effective in complex cyber-infrastructures than trying to control every aspect of a system with centralized ex-ante access control mechanisms. To demonstrate this approach, next section illustrates an example scenario motivated by [171].

6.4 Executable Model of Data Market-Place

In the scenario, there are four actors, *Alice*, *Bob*, *Charlie*, and *David*. From these actors, *David* needs a set of synthesized data (e.g., a trained model) called `final_result`, to create this data set, first the algorithm `alg1` needs to be executed on a data set `data1` and then the output of `alg1` called `result1` needs to be used as the input of the algorithm `alg2` to create `final_result`. The issue is David does not have any of these data or algorithms, instead, Alice has the two algorithms and Bob has the data. The other issue is that Alice and Bob do not trust each other with their data or algorithms (for any reason). However, they both trust Charlie whom also happens to have the computational infrastructure. Figure 6.2 a possible solution application that can be utilized in this scenario. The steps of this application are:

1. In any order:
   (a) Bob sends `data1` to Charlie
   (b) Alice sends `alg1` to Charlie
2. Charlie executes `alg1` on `data1` to create `result1`
3. Charlie sends `result1` to Alice
4. Alice executes `alg2` on `result1` to create `final_result`
5. Alice sends `final_result` to David

```
Act send_data
  Actor source Recipient target
  Related to data
  Terminates to_send_data()

Act compute
  Actor source Recipient target
  Related to input, algorithm, reference
  Terminates to_compute()

Duty to_send_data
  Holder source  Claimant target
  Related to data

Duty to_compute
  Holder source Claimant target
  Related to input, algorithm, reference

Event init_contract
```

Listing 19: Generic data-sharing contract notions in eFLINT

6.4.1 Implementation of the Model

To implement a model of the system, the idea of normative advisors in Chapter 5 is utilized, using ASC2 agents to model the participants of the DMP, and eFLINT programs to model the contractual agreements of the DMP.

Contractual Agreements

To implement the norms, we first start with a drastically simplified of the general eFLINT specification for concepts in the data-sharing contract presented in Listing 19. The specification defines two acts `send_data` and `compute` that respectively represent the act of sending a data object (or algorithm) and the act of computing a result based on input data and an algorithm. There are also two duties that correspond to performing the mentioned acts. Note that performance of each act also terminates the corresponding duty. There is also an event defined as `init_contract` that signals the start of a data-sharing contract. Note that the specification is very minimal and generic and not immediately usable, acts do not specify any side effects except terminating the corresponding duties, also the `init_contract` event as it is defined has zero effect on the institutional state. This is because this specification is only there to define concepts, and each agent at run-time will get a *specialized* –filled in or concretized– version of this specification.

To implement the fully distributed control mechanism, each agent will have its own normative advisor. It is also the goal for each agent to have only access to its own duties and powers, plus the duties and powers of other agents when they

are supposed to fully observe them to act as a monitoring point. Building on the specification in Listing 19, each agent at run-time gets a normative advisor with a specialized *extended* contract specification embedded within it. We start with the simplest specializations, which are David and Bob. Observing the scenario, Bob has only one duty and act to perform –sending `data1` to Charlie – and no expectations from others while David has no duty or acts to perform while only having one expectation –Alice sending `final_result`– from other agents. The two Listings 20 and 21 respectively present the extended specification for Bob and David.

Listing 20 illustrates Bob's specialized contract. Line 1 instructs eFLINT's reasoner to include the data-sharing basics specification (Listing 19). Then, line 3 extends the `init_contract` event, so that at the start of the contract Bob has minimal required information about the institutional state of the market-place, i.e., that it has a data set called `data1`, there are two parties Bob and Charlie –from the perspective of Bob– denoted as `B` and `C`, and that Bob has a power and a duty to send `data1` to Charlie.

```
1  #require "data_sharing_basics.eflint".
2
3  Extend Event init_contract
4    Creates
5      data("data1"),
6      send_data("B","C","data1"),
7      to_send_data("B","C","data1"),
8      party("B"), party("C").
```

Listing 20: Bob's data-sharing contract in eFLINT

Similar to Bob, David also has a simple extension to its contract specification, presented in Listing 21. It extends the contract initialisation event so that it defines two parties, Alice and David (`A` and `D`), and also creates a power and a duty for Alice to send `final_result` to David. As the actor/holder of this power/duty is Alice, from the perspective of David it becomes an expectation that David has from Alice, making him a local monitor for this act.

```
1  Extend Event init_contract
2    Creates
3      send_data("A","D","final_result"),
4      to_send_data("A","D","final_result"),
5      party("A"), party("D").
```

Listing 21: David's data-sharing contract in eFLINT

Next is Charlie, with the role to get an algorithm and a data set from two other parties, perform the computation and return the results to one of them. An

excerpt of Charlie's contract extension is presented in Listing 22. For Charlie, the contract initialization is extended (Line 20) so that it results in creation of two sets of powers/duties: one for Alice to send the algorithm to Charlie and another for Bob to send the data; both of which are expectations from the perspective of Charlie. Then, Charlie's specification also extends the computing act and duty so that they are **conditionally** created when the initially expected `send_data` are performed (Lines 11 and 15) –more specifically when `data1` and `alg1` are available–. The compute act is also extended so that when it is performed with the correct parameters, it creates `result1` (Line 7). Finally, the power and duty to send `result1` to Alice is **conditionally** created when computation is done (Lines 1 and 4) –again, more specifically when `result1` is available–.

```
1  Extend Act send_data
2    Holds when source == "C" && target == "A" && data == "result1".
3
4  Extend Duty to_send_data
5    Holds when source == "C" && target == "A" && data == "result1".
6
7  Extend Act compute
8    Creates data("result1")
9      When input && algorithm && input == "data1" && algorithm == "alg1".
10
11 Extend Act compute Holds when source == "C" && target == "A" &&
12   input == "data1" && algorithm == "alg1" &&
13   reference == "result1".
14
15 Extend Duty to_compute
16   Holds when source == "C" && target == "A" &&
17   input == "data1" && algorithm == "alg1" &&
18   reference == "result1".
19
20 Extend Event init_contract
21   Creates
22     send_data("A","C","alg1"), send_data("B","C","data1"),
23     to_send_data("A","C","alg1"), to_send_data("B","C","data1"),
24     party("A"), party("B"), party("C"),
25     reference("result1").
```

Listing 22: Charlie's data-sharing contract in eFLINT

Finally, Listing 23 shows an excerpt of Alice's contract specialization. Contract initialization is extended (Line 13) so a power/duty is created for Alice to send `alg1` to Charlie. However, this specification also extends the `send_data` act (Line 4) so that when Alice sends `alg1` to Charlie, an expectation for receiving `result1` from Charlie is created. The rest of Alice's specification is similar to Charlie: power/duty to compute `final_result` is created when `result1` is received (omitted Lines 9-11) and power/duty to send `final_result` to David is created when it is

available after computation (omitted Lines 1-2).

```
1  Extend Act send_data ...
2  Extend Duty to_send_data ...
3
4  Extend Act send_data
5    Creates
6      send_data("C","A","result1"), to_send_data("C","A","result1")
7        When source == "A" && target == "C" && data == "alg1".
8
9  Extend Act compute ...
10 Extend Act compute ...
11 Extend Duty to_compute ...
12
13 Extend Event init_contract
14   Creates data("alg1"), data("alg2"),
15   send_data("A","C","alg1"),to_send_data("A","C","alg1"),
16   party("A"), party("C"), party("D"),
17   reference("final_result").
```

Listing 23: Alice's data-sharing contract in eFLINT

Market Participants

To fully implement the model of the DMP, after defining the contract specifications for each actor, scripts for intentional agents and normative advisors needs to be created. As an effect of using dynamic norm specifications and normative advisors, we can implement the agent script and normative advisor script in a way that will be usable for all 4 roles. This is immensely important because it means a much higher usability and as a result scalability of development cycle for the models. In short, to define new types of interactions and scenarios, no modification to the agent scripts is required by the modeller. Instead, they can just define new contracts to create scenarios with any number of agents.

Listing 24 illustrates parts of the ASC2 script for the DMP advisors agent[2]. The two plans that are triggered when a duty `to_send_data` is created or terminated in the embedded eFLINT reasoner. The advisor then simply relays this update to its parent, which is a participant agent in the DMP.

For a final piece of the model, Listing 25 illustrates the script for participant agents that act in the DMP. The first plan (Line 1) is for initiation of the contract, which simply informs the advisor about this event. There are two plans for actual data/algorithm level communications between agents; one for sending a data package (Line 3) and another for receiving a package (Line 7). In both cases, the agent also informs the advisor about this. The next three plans are triggered by

[2]More generic parts that have been already presented in Listing 17 are omitted.

```
1  +to_send_data(party(S),party(T),data(D)) =>
2      #tell(Parent,send_data(S,T,D)).
3
4  -to_send_data(party(S),party(T),data(D)) =>
5      #untell(Parent,send_data(S,T,D)).
```

Listing 24: ASC2 specification of DMP participant's norm advisor.

the advisor: to inform the agent it has a duty to send some data package to a target (Line 10), to inform the agent it should expect to observe a transfer of data, and to inform the agent that a duty to send data has been terminated due to some observed event.

```
1  +!init(Advisor) => !inform_advisor(event,init_contract).
2
3  +!send_data(Target,Data) =>
4      #tell(Target,package(Data));
5      !inform_advisor(act,send_data(Self,Target,Data)).
6
7  +package(Data) =>
8      !inform_advisor(Act,send_data(Source,Self,Data)).
9
10 +send_data(S,T,D) : Self == S =>
11     #println("Duty created: " + send_data(S,T,D)).
12     !send_data(T,D).
13
14 +send_data(S,T,D) =>
15     #println("Duty created: " + send_data(S,T,D)).
16
17 -send_data(S,T,D) =>
18     #println("Duty terminated: " + send_data(S,T,D)).
```

Listing 25: ASC2 specification of a DMP participant

6.5 Model Execution and Discussion

Utilizing the illustrated scripts, an executable model of the DMP can be created. This is achieved by the method presented in Chapter 4. For each scenario, a test suite is created with the required type and number of agents. Each market participant agent is created with the script in Listing 25 and then for each participant a normative advisor is created with the script in Listing 24 with the

corresponding eFLINT specification embedded in it. Then, at the start of the test, an event `+!init(Advisor)` is sent to each market participant agent where the `Advisor` variable is filled with the name of the corresponding normative advisor agent. Because of the design of ASC2, we can modify different parts of agents by injecting dependencies. With the specific injected communication layer to the agents, we can generate different types of diagrams based on the execution of the system filtered by type of messages, type of agent roles or even just include one specific agent's perspective. In the following, some example insights and artifacts created by execution of the model are presented.

Firstly, Figure 6.3 illustrates a sequence diagram generated by including only the communications between participant agents (excluding advisors). Intuitively, this generated diagram is a high-level view of the simple scenario contract and can be used as a design artifact for a real system. However, as our MAS is scalable to virtually any number of agents, when the system is bigger, this type of artifact generation can be a valuable asset for the designers.

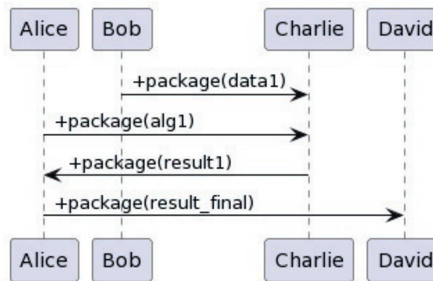

Figure 6.3: Sequence diagram of data-package transfers in the DMP scenario

Next, we will focus further on the normative advisors and their interactions with the agents. For this reason, we choose Charlie as the example case. Figure 6.4 illustrates a sequence diagram consisting of all of Charlie's communications in the scenario. Charlie initially starts by telling the advisor that the contract started, then the advisor informs Charlie that it should expect two data/algorithm packages from Bob and Alice. When each package arrives, Charlie informs the advisor about it and the advisor tells Charlie that it should no longer expect them, however, after both packages it also informs Charlie that about a new duty towards Alice for a computation. After performing the computation, Charlie informs the advisor about it, and in response, the advisor informs Charlie about the termination of the computation duty and creation of a new duty towards Alice to send the results. After sending the results and informing its advisor, the advisor updates the internal institutional state and informs Charlie about the termination of the duty. As it can be seen, while the external communications of this agent are rather simple, the internal institutional state of the agent is changing rapidly,

which in many real world cases is true: even if a behavioral pattern is simple, the underlying interactions that result in those behaviors are complex, in this case the underlying interactions are a legal view over a contractual agreement.

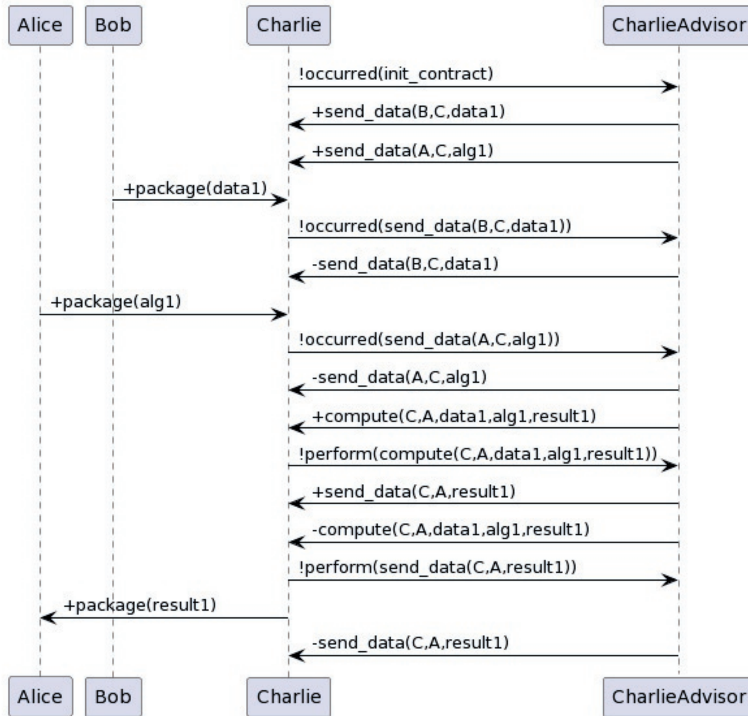

Figure 6.4: Sequence diagram of Charlie's communications the DMP scenario

Finally, we will focus on the concept of local monitoring, deviations between the institutional beliefs of agents and how to identify and possibly synchronize them. The example in Chapter 5 presented agents that have identical norms and also identical observations. This meant that the agents had identical beliefs about the institutional state. However, in the DMP example, this is not the case. While the agents have the same basic concepts in align, each has a specific concretization of norms that it follows. Also, each agent can only observe what is locally available to it: the bilateral communications that it is part of, plus any internal actions that it performs. This has a big effect on accountability in the system, role of trust, need for monitoring and possibility of delegating tasks.

Based on each agent's communications with its advisor, we can create the diagram in Figure 6.5. This figure shows the discrete life-cycle of duties from the perspective of each agent in the environment. On the bottom there are different

events that happen in the system (data packages) which mark the changes in agent's belief about these duties. As it can be seen, these beliefs are not synchronized between agents. For example, David expects to receive `final_result` from Alice from the moment the contract is initialized. However, Alice has a different view over this and believes this duty only exists when Charlie has sent `result1`.

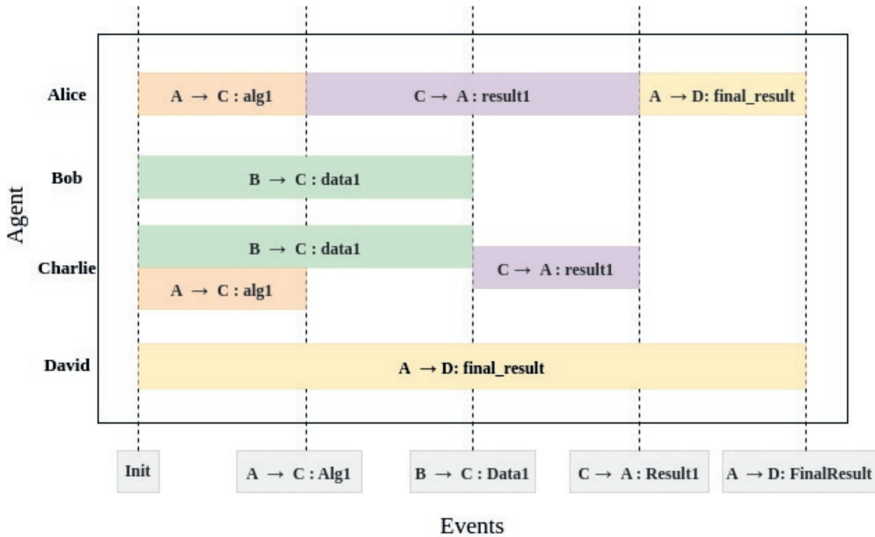

Figure 6.5: Discrete life-cycle of duties, from the perspective of each agent.

This type of deviation in itself is neither desirable nor undesirable, however, in a complex system it is important that designers are aware of it and can study it. For example, imagine that Bob fails to send `data1` to Charlie, and then Charlie fails to send `result1` to Alice, which means Alice can not compute `final_result` for David. Then who should be held accountable from the perspective of David? The answer will depend on the type of agreement between parties. If all of the parties have agreed to be accountable for a task only if previous tasks are completed, then more pre-agreement steps [9] and further monitoring or logging is required in case of failure for possible ex-post auditing and enforcement. But, in case of full delegation, one party may accept to be held accountable for all the sub-tasks that it is delegating to external actors that were not originally part of the agreement –if that is allowed by the policies–. In this case for example, Alice could have an agreement with David to deliver `final_results`, regardless of what happens with other agents, which means there is no need for any further monitoring as David is already (legally) protected by the contract in case of failures.

6.6 Conclusion

This chapter explored a more in-depth analysis of models that can be built by utilizing the tools and frameworks that have been introduced in this thesis. As a proof of concept, a Data Market Place was chosen as the system under scrutiny, and a fully distributed model of task allocation was presented in which the market participants each only get a predefined part of a data-sharing agreement specialized for their role that can assist them at run-time to both fulfill their role and coordinate with other participants. To model the participants, the ASC2 framework was used, while eFLINT was used to model the contractual agreements. Normative advisors as introduced in Chapter 5 were utilized as the glue between the institutional world (contract) and extra-institutional world (data infrastructure).

Although this chapter focused on only modelling the system, in effect, it is not a big leap to implement this approach in a real system in the future: creating and deploying dynamic normative micro-services to advise participants in an ad-hoc or centralized manner and provide them with required information about their and others' roles in a distributed application. Automated norm enforcement and ensuring compliance through explicit and formal representation of norms has many benefits, such approaches can assist both policy-makers and system designers by simplifying their view resulting in a more scalable development process, increase trust between parties and reduce risks that arise from (intended or unintended) non-compliant behaviors.

While this chapter did not explore the addition of domain, consortium, market, and national policies and legislation, the approach is fully compatible with including external overarching policies. For example, in [157] shows how data protection rules (e.g., GDPR) and system level policies can be interconnected. However, this is not a simple matter even from a legal perspective and relations between norms at different levels and their priorities and effects can be much more complex [92], and they become even more complicated in the context of a cyber-infrastructure that includes large volumes of data, which makes the goals of this thesis, being able to model such interactions much more important.

Chapter 7

Example 2: Qualitative, Quantitative, and Normative Reasoning

This chapter illustrates an example model built utilizing the ASC2 framework and its extensions, including preference reasoning and normative advisors. The example was chosen intentionally to be maximally different from what have been presented until this point in the thesis to elaborate on the diverse applications of the approach. More specifically, the agents in this example have to make decisions based on adopted external norms and internal preferences that are dependent on quantitative valuations in the context of the environment.[1]

7.1 Introduction

This example case aims to study the extent AI systems can operate within applicable legal constraints, focusing on the development of autonomous military devices constrained by design by International Humanitarian Law (IHL).[2] This is a relevant use case also because there is a contemporary research debate in which some scholars argue that incorporating many principles of IHL, such as distinction, proportionality, and precautions, into an AI is impossible [51, 154], while some other commentators point out that not only it is possible, but it may be a desirable approach as a well-functioning military AI can possibly provide better performance and increased respect for the law [59, 103].

[1]The material presented in this chapter refines and extends elements presented in [178]. The contribution of the author to this paper has been mainly in the encoding and implementing the norms, designing agent scripts, agent preferences, and the scenarios. The original model of International Humanitarian Law (IHL) is formally developed by the other co-authors Zurek and Kwik [178], as part of the DILEMA (`https://www.asser.nl/dilema/`) project.

[2]Development and utilization of autonomous military devices, fully or partially controlled by artificial intelligence is a controversial idea, charged with moral, legal, and ethical issues, which this thesis does not aim, nor try to address.

This chapter is both a summary and an extension (focused more on the modelling approach rather than the system and laws being modelled) of the work presented in [178], and introduces a formal encoding of IHL rules and their implementation with the use of ASC2 and eFLINT/Prolog languages. The general structure and model of a decision-making mechanism for an IHL-compliant military autonomous devices that the models in this chapter are based on were first introduced in [109].

7.2 International Humanitarian Law rules

International humanitarian law is the set of rules governing all military operations, including weapons release [80]. These principles include guaranteeing that the weapon is sufficiently accurate so as to not be indiscriminate, that attacks are proportionate, and all necessary precautions are taken to spare the civilian population. Such legal requirements must be complied with even if some or all of these decisions are delegated to autonomous devices, and commanders envisaging the use of such devices must ensure that these systems can perform all the required legal tests correctly [23].

Main IHL principles studied in this section are related to targeting and weaponeering, which are implemented through a series of legal tests during the targeting process [72, 165, 47]. The authors of [109] structured and streamlined these legal tests for implementation by a hypothetical military autonomous device. In [178], the discussion is limited to the implementation of tests which are commonly described as the most difficult tasks for an artificial agent to perform, namely those which involve the incidental harm (IH) and military advantage (MA) variables [23]. The tests in question are the *proportionality rule* and the two *civilian harm minimisation rules*,[3].

7.3 The Model

The basis of the model is in the analysis of various relations between MA and IH which respectively are expressed by two values: v_{MA} representing Military Advantage and v_{CIV} representing civilian well-being. For better expressivity, the value v_{CIV} is used which is inversely proportional to IH, $v_{CIV} = 1/v_{IH}$.

The model within the agents can be expressed as $< D, V, p >$, where $D = \{d_x, d_y, ...\}$ represents the available (attack) decisions for the autonomous device. [4] In this definition, V is the set of evaluations of the results of decisions as

[3] Articles 57(2)(a)(iii), 57(2)(a)(ii) and 57(3) of [2] respectively.

[4] How these decisions have been distinguished and represented (e.g. they can be seen as BDI goals) are not further examined here as it will not affect the rest of the model which is the focus in this chapter. It is also assumed that the certainty of the outcomes of those decisions (e.g., destruction of a given tank or bridge nearby) can be predicted.

$V = \{V(d_x), V(d_y), ...\}$ where each evaluation is expressed with two values in the form of $V(d) = \{v_{CIV}(d),\ v_{MA}(d)\}$. Every evaluation is expressed by a real number between -1 and +1 (e.g.: $v_{CIV}(d_x) = 0,75$) which represents the expected satisfaction[5] of the respective variable as the result of the decision. Finally, p is the proportionality coefficient, a real number declared in advance, represents the level of acceptable (from the point of view of IHL) relations between military advantage and incidental harm to fulfil the Proportionality test.

Next, we introduce four different definitions that are necessary for legal tests based on v_{MA} and v_{CIV}:

Definition 7.1 (Equal Military Advantage Predicate (EQMA))
The value $EQMA(d_x, d_y)$ defines whether two different decisions satisfy v_{MA} to the same level. If by d_x and d_y we denote two different decisions, then by $EQMA(d_x, d_y)$ we denote that both decisions satisfy MA to the same level.

$$(v_{MA}(d_x) = v_{MA}(d_y)) \Rightarrow EQMA(d_x, d_y)$$

Definition 7.2 (Less Civilian Protection Predicate (LESSCIV))
The value $LESSCIV(d_x, d_y)$ defines whether one of two decisions satisfy v_{CIV} to a greater extent than the other. If by d_x and d_y denote two different decisions, then $LESSCIV(d_x, d_y)$ denotes that d_x satisfies value v_{CIV} to a lower extent than d_y.

$$v_{CIV}(d_x) < v_{CIV}(d_y) \Rightarrow LESSCIV(d_x, d_y)$$

Definition 7.3 (Proportionality Predicate (PROP))
The value $PROP(d_x)$ defines whether the level of satisfaction of the well-being of civilians (v_{CIV}) by results of a given decision, multiplied by a certain coefficient, is higher than the level of satisfaction of military advantage ($v_M A$) by the same decision. In other words, defines whether military advantage is proportionate to a change in the well-being of civilians.

$$v_{MA}(d_x) \leq p * v_{Civ}(d_x) \Rightarrow PROP(d_x)$$

Definition 7.4 (More Relative Advantage Predicate (MOREREL))
The value $MOREREL(d_x, d_y)$ defines whether the relative satisfaction of MA and IH by one decision is higher than another one. Then $MOREREL(d_x, d_y)$ denotes that the relation of MA to IH for d_x is higher than it is for d_y.

$$v_{MA}(d_x) * v_{Civ}(d_x) \geq v_{MA}(d_y) * v_{Civ}(d_y) \Rightarrow MOREREL(d_x, d_y)$$

Finally, based on these predicates a set of legal rules representing tests necessary to examine whether a given decision is legal from the point of view of IHL are introduced:

[5] In the actual implementation of the model each decision can have multiple possible outcomes with different probabilities, and the expected satisfaction is calculated based on those values.

Definition 7.5 (Article 57(3) test)
If more than one target is viable, and they produce comparable MA, then the target with the lowest IH should be selected. The predicate $DT(d_x)$ represents this legal test, where d_x is the decision which satisfies the test:

$$\exists_{d_x \in D} \neg \exists_{d_y \in D} (EQMA(d_x, d_y) \wedge LESSCIV(d_x, d_y)) \Rightarrow DT(d_x)$$

Definition 7.6 (Proportionality test)
The predicate $DP(d_x)$ defines that decision d_x is proportional with respect to its military advantage and incidental harm.

$$PROP(d_x) \Rightarrow DP(d_x)$$

Definition 7.7 (Minimisation of incidental harm)
Predicate $DMH(d_x)$ defines that a decision d_x passes the minimisation test with respect to the incidental harm it will cause.

$$\exists_{d_x \in D} \forall_{d_y \in D} (MOREREL(d_x, d_y)) \Rightarrow DMH(d_x)$$

A given targeting decision will be coherent with IHL if all the above tests will be fulfilled, therefore on the basis of all the defined earlier predicates we can create a rule describing whether a given decision will follow IHL.

Definition 7.8 (Rule of IHL)
The predicate $DAV(d_x)$ denotes that a decision d_x fulfills the requirements of being a legal decision in accordance to IHL if we have $DT(d_x)$, $DP(d_x)$, and $DMH(d_x)$.

$$DT(d_x) \wedge DP(d_x) \wedge DMH(d_x) \Rightarrow DAV(d_x)$$

7.4 Decision-making

The decision-making of the military device can be any arbitrary mechanism as long as the final decision d_x passes all the tests meaning we have $DAV(d_x)$. For example, the decision-making mechanism can choose, from the set of available decisions which fulfil IHL rules, the one which brings about the highest military advantage. Let $Decisions = (D, \geq)$ be a total ordered set representing ranking of decisions. The basis of this ordering is military advantage, assuming that commanders would prefer decisions which provide the greatest expected military utility from all lawful alternatives:

$$\forall_{d_x, d_y \in D} (DAV(d_x) \wedge DAV(d_y) \wedge (v_{MA}(d_x) \geq v_{MA}(d_y)) \Rightarrow (d_x \geq d_y) \tag{7.1}$$

This means that if the set DAV is empty (no decision remains), then this means that there is no possibility to make an attack which satisfies the set military goals and which is also coherent with IHL. If there is one decision satisfying the

requirements only, this decision becomes the final one. If more than one decision satisfy the requirements, they are ordered on the basis of expected military advantage. The system, on the basis of the ordering *Decisions* can choose the best decision (the action bringing about the highest MA) and follow that decision (fulfill the plan).

7.5 Example Model of Normative Autonomous Devices

In this section, we explore scenarios including models of imaginary autonomous military devices (drones) that follow the aforementioned IHL principles in action. Then, a simple implementation of the devices in the scenarios are presented that utilize ASC2 and eFLINT.

7.5.1 Scenario

We present below the overarching scenario on the basis of which we are going to test our mechanism:

> A commander from nation Alpha is given the task to capture a city defended by nation Beta, which uses the city's smart sensors to collect data of Alpha's troop movements and plan effective counterattacks. For each district (scenario), data is collected at a data center before being sent through relay stations to Beta's headquarters. Aiming to disrupt Beta's intelligence network, Alpha's commander releases *Cleopatra* drones which are given the task to locate the data centers or relay stations ('network points') and destroy one of them, which would disable the data flow in that district. Network points can be located inside civilian buildings, on rooftops or in fields. The drones are able to identify civilians and enemy soldiers around potential target locations and take this information into consideration for their decision-making. *Cleopatras* carry two types of ammunition, 'light' and 'heavy' missiles. Heavy missiles are necessary for attacking targets inside buildings, but cause more damage to their surroundings. The risk of misidentification or released missiles missing the target is negligible.

On the basis of the above scenario we assume that a particular drone in a given situation can make a decision concerning destroying one of the network points namely relay stations (RelSta) or data-centers (DCenter), with one of two different kinds of missiles (heavy and light), giving *2n* possible decisions to examine. In order to examine the mechanism, we assumed four sub-scenarios with different collateral effects that can be predicted by the device (see Table 7.1). Each table

presents a different district or scenario (A, B, C, and D) where each row presents a different attack decision. Each decision, apart from the target and missile type, shows the location which is relevant as only heavy missiles are effective against buildings. The number of enemy soldiers neutralized as a side-effect of the attack is illustrated in the 'Sldrs', the number of collateral civilian lives and buildings are also presented in the columns 'CL' and 'CB'. The corresponding v_{MA} and v_{CIV} values are calculated based on these parameters. The military advantage of all targets (network devices) is the same, given the missile type is viable in that location. However, there is a variation of v_{MA} in each row based on the number of enemy soldiers. The v_{CIV} value is based on the number of civilian buildings and lives. Both values are also affected by a random modifier and rounding in each decision.

The full analysis of the operations in each district is presented later on, but in summary to rationalize four different scenarios, **District A** is a generic scenario where there are not a lot of civilians, however, only one option is legal. In **District B**, there are many civilians around and all options will result in high collateral damages. In **District C**, an extremely high-value target is present (Beta's president, indicated by 'P') which can be neutralized with a heavy missile at the cost of many civilian casualties. Finally, in **District D**, like **A**, there are not many civilians around, but more than one option is legal, so it will be agent's decision-making that needs to select one.

7.5.2 Implementation

This section presents the basics of the implementation of the experiment. The proof of concept is implemented with the approach presented in 5 with two components: (1) an intentional agent that encapsulates the objectives and procedural knowledge that is implemented utilizing ASC2 framework and (2) a normative advisor that encompasses the normative aspects. Using intentional agents and normative advisors is advantageous in this case because of the separation of the analysis of legality of the decision from making the decision itself. Such a separation is important because it preserves the required level of transparency concerning the IHL compliance: in particular, it allows for clear understanding why a given decision fulfills a particular IHL rule.

The IHL rules in normative advisors are implemented twice with two languages, once with eFLINT and once with Prolog. The choice to use Prolog was taken in the process of implementation, because it turned out that eFLINT's current version is not optimized for this specific encoding which resulted in low performance. However, as the rules are already encoded in logical form, Prolog is an intuitive choice. This extra step is explicitly presented in this chapter to illustrate the versatility and modularity of the normative advisors and how they are agnostic towards specific the norms framework.

```
% Beliefs, Rules
viable(D) :- pass_ihl_rules(D).

% Preferences
+!engage_viable_target(D1) >> +!engage_viable_target(D2) :-
    target(D1) >> target(D2).
target(D1) >> target(D2) :-
    evma(D1,V1) && evma(D2,V2) && V1 > V2.

% Plans
% P1
+!engage() =>
    !engage_target(D).
% P2
+!engage_target(D) =>
    !engage_viable_target(D).

% Internal Plans
% P3
@internal @preferences
+!engage_viable_target(D)
    : target(T) && viable(D)
        => #log("targeting: " + D);
          #initiate_attack(D).
% P4
@internal
+!engage_viable_target(D)
    : not viable(D)
        => #log("Not a viable target: " + D).

% Sync data with advisor
+data(Fact) => #tell("IHLAdvisor", Fact).
-data(Fact) => #untell("IHLAdvisor", Fact).
```

Listing 26: ASC2 implementation of IHL compliant device

Table 7.1: Districts A, B, C, and D: Sample decision lists

District	Dcsn	Target	Location	Sldrs	Msl type	CB	CL	v_{MA}	v_{CIV}
A	1	RelSta1	roof	0	heavy	1	6	0.5	0.4
A	2	RelSta1	roof	0	light	1	2	0.5	0.6
A	3	RelSta2	field	5	heavy	0	5	0.6	0.7
A	**4**	**RelSta2**	**field**	**5**	**light**	**0**	**2**	**0.6**	**0.8**
A	5	DCenter	building	5	heavy	2	10	0.6	0.2
A	6	DCenter	building	5	light	1	4	0.05	0.5
B	1	RelSta1	roof	0	heavy	3	10	0.5	0.15
B	2	RelSta1	roof	0	light	1	6	0.5	0.4
B	3	RelSta2	building	0	heavy	3	15	0.5	0.1
B	4	RelSta2	building	0	light	1	2	0.05	0.6
B	5	DCenter	building	5	heavy	2	10	0.6	0.2
B	6	DCenter	building	5	light	2	4	0.05	0.5
C	1	RelSta1	field	0	heavy	0	5	0.5	0.7
C	**2**	**RelSta1**	**field**	**0**	**light**	**0**	**2**	**0.5**	**0.8**
C	3	RelSta2	building	5	heavy	3	15	0.6	0.1
C	4	RelSta2	building	5	light	1	2	0.05	0.6
C	5	DCenter	building	50+P	heavy	4	150	0.95	0.01
C	6	DCenter	building	5	light	1	4	0.05	0.5
D	1	RelSta1	roof	0	heavy	1	6	0.5	0.4
D	**2**	**RelSta1**	**roof**	**5**	**light**	**0**	**4**	**0.6**	**0.75**
D	3	RelSta2	field	5	heavy	0	5	0.6	0.7
D	**4**	**RelSta2**	**field**	**0**	**light**	**0**	**1**	**0.5**	**0.9**
D	5	DCenter	building	5	heavy	2	10	0.6	0.2
D	6	DCenter	building	5	light	1	4	0.05	0.5

Drone (Intentional) Agent

The drone's intentional agent's script implementation as a BDI agent can be seen in Listing 26. The drone has one inference rule (line 2) specifying a target is `viable` if it passes IHL rules defined by predicate `pass_ihl_rules`. The agent itself does not define when a decision passes IHL rules, which keeps it agnostic towards specific rules; instead, this information is fed to it by the advisor. Next, there are two preference statements encoded in CP-Nets that are an encoding of formula 7.1. These preferences are used to make decisions between targets, stating that agent will prefer targets with higher `evma` (lines 4-6). The first statements specify that when the agent is engaging a viable target, "it prefers to engage a more preferred target", this is an example of nested preferences introduced in Chapter 3. The second statement defines that a target with higher `evma` is preferred with no context. Also note that the preference statements do not need to take into account the IHL rules, making them far more modular.

The plans P1 and P2 are the main plans for external events, P1 can be used to respond to the event of achieving `engage`, meaning engage *some* target without specifying any specific one that when committed will simply adopt an abstract

sub-goal of `!engage_target(D)` where the parameter `D` is a free variable; note that this is an example of an abstract goal from Chapter 3 that will be grounded later on. Plan P2 matches with the event `engage_target(D)` where `D` can be either a free variable or a grounded one which will make it a concrete goal; the body of P2 simply has a sub-goal of `!engage_viable_target(D)`.

The two internal[6] plans P3 and P4 are both relevant for the event or goal of `engage_viable_target(D)`. In the case of P3, it is applicable when `viable(D)` is true according to agent's beliefs, and, vice-versa, P4 is relevant when this is not the case. Note that in P3, if the parameter `D` is grounded then it is checked if `viable(D)` can be proven, however, if `D` is a free variable, the agent tries to find the most preferred substitution for `D` in its beliefs that `viable(D)` is true for, the most preferred here is determined by the preference rules in lines 4-6 as the one with the highest `evma`. This means the agent can handle the two main objectives of finding a viable target, or, checking if an already specified target is viable. The bodies of P3 and P4 are intuitive, P3 initiates an attack and P4 just logs the failure. Intuitively, in a more realistic implementation there should be further plans to work around the failure, e.g., by changing the requirements of what constitutes as a viable target by adding new rules.

The last two plans are simple information relay plans that allow the agent to send any new data (facts from the normative perspective) to the advisor to keep its information up-to-date with the agent's observations. Note that while here there are only simple one-to-one synchronization plans, this process can be extended with any arbitrary qualification process that maps the observations of the agent into normative facts.

eFLINT Powered Normative Advisor

IHL in eFLINT Listing 27 shows the excerpt of an eFLINT specification for our running example. This specification shows only a small subset of what eFLINT can encode. The script instead defines multiple types of facts, some atomic ones like `target` and `vma`, some composite ones like `outcome` and the rest are *derived* facts. Some examples are: In line 5 the fact `evciv(target,value)` which derives the expected value of civilian well-being for a target from all the possible outcomes of that target. In line 11 the fact `proportionate(target)` which is derived from the proportionality formula (Definition 7.3). Fact `dp(target)` in line 18 defines proportionality test (Definition 7.6), fact `dt(target)` in line 21 defines Article 57(3) test (Definition 7.5), fact `dmh(target)` defines the harm minimization test (Definition 7.7), and finally in line 29, fact `dav(target)` holds when four facts `dp`, `dt`, `dmh`, and, `dtr` holds for that target (Definition 7.8). Note that eFLINT by design includes a transition system that on every update proactively searches for all the possible facts (or acts, or duties) that can be derived and creates them and

[6]Internal means that the agent will not adopt these goals as an external event.

```
// Composite Fact Types
Fact outcome Identified by target * vciv * vma * probability
...
// Derived and Inferred Facts
Fact evciv Identified by target * value
  Derived from evciv(target,
    value(Sum( Foreach outcome :
      (outcome.vciv * outcome.probability) When
        outcome.target == target) / 100 ))
...
Fact proportionate Identified by target
  Holds when
     evciv(target,value) &&
     evma(target,other-value) &&
     proportionality-coefficient(coeff-value) &&
     other-value <= ((value * coeff-value) / 100)
...
Fact dp Identified by target
  Holds when proportionate(target)

Fact dt Identified by target
  Holds when !(Exists other_target : other_target != target &&
  eqma(target,other_target) && lessciv(target,other_target))

Fact dmh Identified by target
  Holds when !(Exists other_target : other_target != target &&
  !morerel(target,other_target))
...
Fact dav Identified by target
  Holds when
    dp(target) && dt(target) && dmh(target) && dtr(target)
```

Listing 27: Excerpt of IHL encoded in eFLINT DSL

then as it will be shown in the following they are reported to the advisor agent.

Advisor in ASC2 The normative advisor agent script specialized for IHL rules implemented in eFLINT (Listing 27) is illustrated in Listing 28, the first six plans (lines 1-8) can be utilized to communicate with the eFLINT instance within the agent to check if an act is enabled, to perform an act, or to check if a fact holds true. Although these are enough for the intentional part of the device to check if a target passes all the tests – e.g., by querying `?holds(dav(D))`–, we will take a more proactive approach. The last 4 plans (lines 10-13) illustrate this, they are plans that tell the normative advisor to report specific updates within the normative state of the environment back to the device, namely assertion and retraction of facts `dav` and `evma`. Furthermore, the plans in lines 10 and 11 also have an extra qualification step relaying that the fact `dav` *counts as* the fact `pass_ihl_rules`.

This allows for modular design as the intentional part of the device does not need to know the specific rules in IHL, meaning even if IHL rules are changed, only this qualification rule needs to be updated and not the intentional agent.

```
1  +?permitted(A): enabled(A) => #respond(true).
2  +?permitted(A) => #respond(false).
3  
4  +!perform(A): enabled(A) =>  #perform(A).
5  +!perform(A) => #coms.inform(Source, failed(A)).
6  
7  +?holds(A): holds(A) => #coms.respond(true).
8  +?holds(A) => #coms.respond(false).
9  
10 +dav(D) =>   #tell("Device", pass_ihl_rules(D)).
11 -dav(D) => #untell("Device", pass_ihl_rules(D)).
12 +evma(D,V) =>   #tell("Device",evma(D,V)).
13 -evma(D,V) => #untell("Device",evma(D,V)).
```

Listing 28: ASC2 implementation of IHL eFLINT Powered advisors

Pure ASC2 Advisor

The alternative approach to create the IHL advisor is to use pure ASC2 script which is illustrated partially in Listing 29. This is possible because ASC2 agents already have an internal Prolog engine embedded in them. Alike to the eFLINT specification, lines 6, 13, 15, 21, and 25 respectively represent proportionality formula (Definition 7.3), proportionality test (Definition 7.6), Article 57(3) test (Definition 7.5), harm minimization test (Definition 7.7), and finally, overall passing of all other tests (Definition 7.8). The main difference from the eFLINT implementation is that unlike before, Prolog fact are not proactively analyzed and queries should be triggered externally, in this case the advisor agent has two plans (lines 28 and 32) that when an outcome for a target (decision) is asserted or retracted. As a result, the agent will then adopt two goals to firstly update the intentional agent about newly `evciv` and `evma` values for that target (lines 36-40), and then also determine if this target passes all the tests (or not) simply by checking if the query `dav(D)` holds (or not) according to its belief base, and relay the result to the intentional agent (lines 42-43 and lines 45-46).

7.6 Model Execution and Discussion

To run the experiments, approaches detailed in Chapter 4 are utilized, where each scenario is implemented in a test suite; the full implementation of the agents

```
% Rules
evciv(D,V) :-
    findall(VCIV*P,outcome(D,VCIV,VMA,P),VCIVLIST) &&
    sumlist(VCIVLIST,V).
...
proportionate(D) :-
    evma(D,EVMA) &&
    evciv(D,EVCIV) &&
    prp(Prp) &&
    PEVCIV is EVCIV * Prp &&
    EVMA =< EVCIV.
...
dp(D) :- target(D) && prop(D).

dt(D1) :-
    target(D1) &&
    forall(
      target(D2),
      (D1 !== D2 && eqma(D1,D2) && lessciv(D1,D2)) -> false || true).

dmh(D1) :-
    target(D1) &&
    forall(target(D2), (D1 !== D2 && not morerel(D1,D2)) -> false || true).

dav(D) :- dt(D) && dp(D) && dmh(D).

% Plans
+outcome(D,_,_,_) : target(D) =>
    !update_values(D);
    !update_dav(D).

-outcome(D,_,_,_) : target(D) =>
    !update_values(D);
    !update_dav(D).

+!update_values(D) : evciv(D,EVCIV) && evma(D,EVMA) =>
    #coms.untell("IHLDevice",evciv(D,_));
    #coms.untell("IHLDevice",evma(D,_));
    #coms.tell("IHLDevice",evciv(D,EVCIV));
    #coms.tell("IHLDevice",evma(D,EVMA)).

+!update_dav(D) : dav(D) =>
    #coms.tell("IHLDevice",pass_ihl_rules(D)).

+!update_dav(D) : not dav(D) =>
    #coms.untell("IHLDevice",pass_ihl_rules(D)).
```

Listing 29: ASC2 implementation of IHL in Prolog Powered Advisors

and test scenarios is publicly available online[7]. In each case, an instance of the intentional agent plus an instance of an advisor agent is created. Then, a list of available decisions with their evaluations for each district as presented in Table 7.1 is sent to the intentional agent sequentially. After the last decision is sent, the system is triggered to perform the decision-making. Then, the agent inspects the norms instance embedded in the advisor to see which facts are present. In the example, each decision is identified by the target name (e.g., RelSta1) and the missile type (e.g., heavy) and a proportionality factor of 1 is used ($p = 1$).

7.6.1 Execution Results

The results of the IHL compliance analysis are presented in Table 7.2, each district or scenario in a separate sub-table. In the following, each scenario is analysed by heightening its interesting points.

District A In District A, only decision number 4 satisfies all of the requirements. Decisions 1 and 5 are not proportional as their ev_{MA} outweighs their ev_{CIV}, while decisions 1, 3, and 5 do not meet the Article 57(3) test requirement because for each of them there is at least another decision with the same ev_{MA} but a higher ev_{CIV}. Finally, decisions 2, 6 do not meet the minimization of incidental harm requirement because there is another available decision, namely 4 that has a higher relative satisfaction of military advantage and incidental harm, meaning only decision 4 is compliant to IHL.

District B In District B, there are no compliant decisions, what makes this scenario interesting is the interaction between decisions 2 and 4. Decision 4 does not satisfy the minimization of incidental harm rule, as there exists another decision, namely 2, that has a higher relative satisfaction of military advantage and incidental harm. However, decision 2 is not proportional itself, meaning no compliant decisions exist in this scenario.

District C In District C, we have decision 5 that results in a very high ev_{MA} and a very low ev_{CIV}, in effect this is an operation that brings about a lot of military advantage but a high cost. Intuitively, this decision is not satisfying proportionality and minimization of incidental harm requirements, meaning it is not compliant to the IHL rules.

Ditrict D In this district, there are two decisions that satisfy all the requirements, meaning from the perspective of the IHL advisor agent they are both legal. This results in the need for the intentional agent to make a decision. The preference Formula 7.1 as encoded in Listing 26 lines 5-8 is used by the intentional agent to

[7] `https://github.com/mostafamohajeri/jurix2022-ihl-devices`

Table 7.2: Decision Analysis

District	Decision	Target	Msl type	ev_{MA}	ev_{CIV}	DT	DP	DMH	DAV
A	1	RelSta1	heavy	0.5	0.4	✗	✗	✗	✗
A	2	RelSta1	light	0.5	0.6	✓	✓	✗	✗
A	3	RelSta2	heavy	0.6	0.7	✗	✓	✗	✗
A	**4**	**RelSta2**	**light**	**0.6**	**0.8**	✓	✓	✓	✓
A	5	DCenter	heavy	0.6	0.2	✗	✗	✗	✗
A	6	DCenter	light	0.05	0.5	✓	✓	✗	✗
B	1	RelSta1	heavy	0.5	0.15	✗	✗	✗	✗
B	2	RelSta1	light	0.5	0.4	✓	✗	✓	✗
B	3	RelSta2	heavy	0.5	0.1	✗	✗	✗	✗
B	4	RelSta2	light	0.05	0.6	✓	✓	✗	✗
B	5	DCenter	heavy	0.6	0.2	✓	✗	✗	✗
B	6	DCenter	light	0.05	0.5	✗	✓	✗	✗
C	1	RelSta1	heavy	0.5	0.7	✗	✓	✗	✗
C	**2**	**RelSta1**	**light**	**0.5**	**0.8**	✓	✓	✓	✓
C	3	RelSta2	heavy	0.6	0.1	✓	✗	✗	✗
C	4	RelSta2	light	0.05	0.6	✓	✓	✗	✗
C	5	DCenter	heavy	0.95	0.01	✓	✗	✗	✗
C	6	DCenter	light	0.05	0.5	✗	✓	✗	✗
D	1	RelSta1	heavy	0.5	0.4	✗	✓	✗	✗
D	**2**	**RelSta1**	**light**	**0.6**	**0.75**	✓	✓	✓	✓
D	3	RelSta2	heavy	0.6	0.7	✗	✓	✗	✗
D	**4**	**RelSta2**	**light**	**0.5**	**0.9**	✓	✓	✓	✓
D	5	DCenter	heavy	0.6	0.2	✗	✓	✗	✗
D	6	DCenter	light	0.05	0.5	✓	✗	✗	✗

achieve this. According to these preference statements, a decision with a higher ev_{MA} shall be selected, which in this case is decision 2.

7.6.2 Discussion

Although our normative reasoning mechanism is relatively simple, the results obtained (even for controversial cases) are correct. Note that although in the case of District A and District C there is one lawful decision only (in District B there no lawful decisions at all), there are no formal or technical restrictions concerning a greater number of lawful decisions as is the case in District D, however, such situations are rather seldom, because they require the same relation between IH and MA for two or more different decision (unless there is some degree of approximation involved where values are considered *almost* the same). The choice amongst available options as it was presented (decisions which are IHL-compliant) is made by a decision mechanism on the basis of the level of v_{MA}.

The problem of balancing was widely discussed in a number of AI and Law papers and legal case-based reasoning in particular. In legal CBR, the objects of comparison are either dimensions (e.g. [15]) or values (e.g. [13, 89]). The key

difference between our model and the existing ones is in the level of abstraction: both V_{MA} and V_{CIV} have a very abstract character, especially in comparison to dimensions like *number of disclosures*, *income*, or *days of absence in a country*. An important difference also lies in the absolute representation of the level of satisfaction of values, whereas in other models of balancing, the levels of values' promotion was represented in a relative way (in a comparison to other decision, state of affairs, etc. e.g. [89, 116]). Moreover, in contrast to many argumentation or legal reasoning models [14, 175], values in our model are not an external element of a reasoning process allowing for solving conflicts between arguments, but they are an element of a legal rule itself. The simplicity of our model, however, shows that the critical point of the reasoning process is not located in the legal reasoning, but in the calculation of the specific relations between v_{MA} and v_{CIV}. Such an observation allows us to derive a more general conclusion: the key difficulty of targeting compliance testing lies not in the legal reasoning and balancing itself, but in the process of evaluating the available options.

In practice, obtaining v_{MA} and v_{CIV} can be seen as a classification or regression task, which can be expressed as assigning numbers (representing v_{MA} and v_{CIV}) to particular decisions (represented by their specific parameters). The key question is whether the creation of such a regression mechanism is feasible at all. Answering this question will be an important topic for future research.

Chapter 8

Conclusions and Future Work

Modelling norm-governed cyber-infrastructures can be done via multiple approaches, investigated in different research communities. In this thesis, however, the focus was on utilizing agent-based programming to model the system as a collection of autonomous components, that are individually modelled as either intentional agents or as software/hardware infrastructural components. In this Chapter, the motivations that drive this research are reiterated. Then, the work presented throughout the previous chapters is summarized. Next, the achievements of the dissertation are reorganized by assessing the present work against the research questions described in the introduction, all in identifying the current limitations. Finally, possible future research directions are highlighted.

8.1 Motivation

Governance refers to processes, and structures through which organizations, societies as a whole, and actors within them are regulated and participate to regulatory and regulative activities. Governance includes mechanisms by which decisions are made, normative relations are defined, authority is exercised, and actions are taken to achieve goals and fulfill responsibilities and expectations.

Policies and policy-making play a crucial role in governance. Policies consist of sets of rules, regulations, and procedures addressing various actors, guiding their decision-making and ultimately their behavior. They outline the objectives, values, and behavioral patterns that need to be followed to achieve desired outcomes, promote consistency, coherence, and effective coordination of individual actors' behavior. Policies can cover various areas such as legal compliance, ethical standards, operational procedures, resource allocation, and risk management.

Policy-making in the data-sharing domain, as well as in the broader software domain, is becoming increasingly important in the contemporary world. As software systems are getting more involved and integrated in societies, there is an increasing need for governance actions to make sure that these systems

are regulated in accordance with agreed societal norms. This entails a need for approaches to make policies for software systems effective. However, there are many issues that challenge governing software systems, including AI. The number of AI systems has rapidly increased over the last few years, and lack of transparency, concerns about privacy, and other legal and ethical issues have increased the need for effective policies to control AI and more generally IT-infrastructures, including data-sharing systems. This dissertation focuses on this issue, exploring challenges and approaches for policy-making addressing social systems that include both social and software actors, and are governed by a set of regulations.

Modelling is one of the central tools utilized in policy-making. Models can give insight about a system and assist in predicting the trajectory of that system in the future. Agent-based modelling in particular is a powerful tool for policy-making, as such models start from defining the behavior of individual actors to infer the high-level emergent behavior of the system as a whole, enabling to integrate aspects related to micro- (individual), meso- (groups and orgainzations) and macro- (society) levels. This is the approach taken into consideration in this dissertation.

8.2 Summary

The thesis can be separated in two main parts. After the introduction, Chapters 2 to 4 mainly revolve around modelling social agents as BDI agent scripts, putting an emphasis on model usability through enhancing scalability, transparency and testability. The second part, in Chapters 5 to 7, is about modelling social norms, specifically addressing the interaction of social agents with norms, and thus covering the policy-making aspect of the dissertation.

In Chapter 2, an agent-based programming framework called AgentScript Cross-Compiler (ASC2) is introduced. This framework is based on the Belief-Desire-Intention model of agency, which is presented in the literature to model a wide range of socio-technical systems. ASC2 uses a language based on AgentSpeak(L)/Jason [139, 26]. While there are multiple well-developed BDI-based frameworks described in literature, the main requirements and advantages of ASC2's design are scalability and usability. AS2 is a cross-compiler that takes as input agent programs developed in a high-level language, and translates them to executable JVM-based programs. This makes ASC2 models virtually as scalable as any JVM program. Furthermore, the frameworks utilizes the Actor model (implemented in Akka), giving it a consistent and robust backbone for concurrency. ASC2's language, albeit heavily influenced by previous works, has unique characteristics; most importantly, JVM languages (Java/Scala/Kotlin) statements are allowed as part of the language, making it a polyglot of a reactive agent language, a prolog-like logic language, and a JVM-like language. From a performance perspective, Chapter 2 presents multiple benchmarks to compare ASC2 with

other frameworks, which shows ASC2 performs better or equal to other existing frameworks.

Chapter 3 explores the idea of adding explicit preferences at the language-level to BDI agent. Utilizing them to allow the agent to refine abstract goals at run-time based on the situation to achieve its desires. Where BDI scripts are typically defined around the *how-to* knowledge (or procedural knowledge) of an agent, Chapter 3 illustrates that adding preferences enhances them with *what-to* knowledge. Given a partially abstract goal, the agent can gradually concretize and achieve these goals based on its preferences. and the context of the environment in which the agent resides. By allowing explicit preferences to an agent script, many decisions that would be opaque (as they happen within the reasoning engine of the agent) become instead explicit and transparent.

To model the preferences, ASC2 utilizes CP-nets, short for "Conditional Preference Networks", a type of graphical model used to represent and reason about qualitative preferences. They are commonly used in the field of artificial intelligence, specifically in preference modeling and decision-making. Chapter 3 also illustrates a novel form to represent CP-Nets alongside an algorithm to find optimal decisions in Prolog; it proves the correctness of the algorithm and shows that the time complexity of the algorithm is on-par with what is presented in [29]. Furthermore, this new form can also be used to express contextually conditioned and parameterized preferences, resulting in more flexibility than pure CP-Theories. To reiterate an example, one can define a preference statement such as "I prefer the place I am already at (to any other place)" that depends completely on the state of the agent in the environment. This extension makes CP-Nets, a model that is classically used for static one-time decisions in a fully known environment, suitable also for agents acting in dynamic environments with limited information.

Chapter 4 studies testing and verification of agent-based models at scale and in real world settings. As the software engineering community already has many advanced and mature tools for testing, Chapter 4 aims at creating interoperability between agent-based modelling frameworks with mainstream software development tools used for automated testing and integration. It illustrates the interfacing of ASC2 with multiple tools such as build tools, unit and integration testing, continuous integration and deployment, and code coverage systems. It furthermore explores how we could present requirements that would allow other frameworks to adopt the same idea.

Chapter 5 introduces a flexible agent architecture for introducing norms in multi-agent systems. The basis of this architecture are normative advisors. In short, a normative advisor is a normal BDI agent, except for the fact that its inference engine –which is typically a logic-based reasoning engine (in our experiments, eFLINT)– can reason with and about norms. With this approach, these advisor agents can be utilized by other agents as an external source to maintain and reason about an institutional perspective (state) based on a specific set of norms. The advisors can be initialized with a particular norm specification and maintain

an institutional perspective on the environment, which is continuously updated at run-time and can be queried at anytime. Both regulative and constitutive norms are taken into account. This chapter creates the basis to allow an agent-based model being utilized in policy-making.

Chapter 6 and 7 presents two illustrative examples of how the proposed approaches and tools in this dissertation can be utilized to model a normative system. Chapter 6 utilizes ASC2 and its normative extension to illustrate, implement and analyze a model of a Data Market-Place as a normative multi-agent system. Norms in this chapter are used as a coordination mechanism between participants of the market to assist policy-makers by providing insights into the effect of policies, and even generating policy and system design artifacts. The market participants in this example use *ad-hoc* contractual agreements and their bounded view of the events occurring in the system to create their own institutional view of the market. By doing so, each participant creates a dynamic set of expectations about what shall happen in the system. This means that, based on the pre-agreed norms and events happening in real time, each actor infers their duties and claim towards other participants and act following these conclusions.

Chapter 7 presents another illustrative example, focusing on particular challenges in modelling normative systems, namely mixing qualitative, quantitative, and normative reasoning. The agents in the example described in this chapter act based on external norms that they have adopted and internal preferences, both of which depend on the quantitative state of the environment. While the scenarios used in the example described in this chapter are quite simple, they illustrate how a combination of different modelling approaches in one framework can allow for more expressive models.

8.3 Research Results

As described in Chapter 1, the overarching research goal underlying this dissertation is *defining approaches, methodologies and tools for policy-making in the data-sharing domain*. Modelling plays an important role in policy-making; scenario simulation via model execution can give insights and possible predictions about the trajectory (or the possible trajectories) of the system. Furthermore, models facilitate stakeholder engagement and transparency about the system, and, in the case of software-based systems, they can even guide or become part of the development process. Among the different types of modelling, agent-based models are especially suitable for policy-making as they define the behavior of individuals to study emergent behaviors of the system as a whole [67]. The main research question can then be reformulated as:

> *How can we model a norm-governed cyber-infrastructural system for the purpose of policy-making?*

To address this question, this dissertation explored which components of these models need to be specified and what are the requirements of these models to make them suitable for policy-making, specifically in data-sharing. The thesis presented a set of tools in the form of an agent programming framework called ASC2. To identify and build up the components of this framework, this problem was broken down into a set of research questions that the work presented in this dissertation aimed to answer:

> **Research Question 1:** *How can we create expressive, scalable and modular models of social agents?*

Although there are many requirements that agent models may need to satisfy, expressivity, scalability, and, modularity are identified to have higher priority in the context of this research.

Expressive Power The main motivation of using agent-based models in policy-making is to build-up the system level behavior from agent behavior to analyse the effect and impact of policies. This requires modelling the individual decision-making process given subjective social norms, individual preferences, goals and desires. Creating such models demands highly expressive agent models in terms of cognitive capabilities. In Chapter 2 the ASC2 agent-programming framework is introduced. ASC2 is based on the BDI model of agency [140] utilizing an agent programming language called AgentScript which is based on AgentSpeak(L) [139]. The BDI agent theory used in this dissertation builds upon Bratman's theory of practical reasoning [31], describing the agent's cognitive state and reasoning process in terms of its *beliefs*, *desires* and *intentions*, attributes usually used to describe human behaviors, making them highly expressive in modelling social agents [78]. Apart from intrinsic attributes of BDI, ASC2 takes extra steps in adding preferences at the language level, abstract goal refinement at the decision-making level in Chapter 3, and adding norm reasoning at the framework level in Chapter 5.

Scalability Often policy-making requires modelling a system with a high number of individuals. This creates a challenge for agent-based modelling, particularly if these models each have a complex cognitive model with higher computational resource demand. Scalability then becomes a first-class requirement, as having complex agents is not beneficial in policy-making if we can only have a few of them in any given scenario. Scalability refers to the ability of the framework to *scale up* by adding more computational resources, typically in a distributed manner. At the execution level, ASC2 utilizes actor-oriented programming via the Akka actor framework. Each agent consists of multiple actors, each with their own role that can communicate through internal messaging, effectively making the agent a modular actor micro-system in itself. This means that not only between

agents, but even within each agent, there may be internal asynchronous execution of concurrent tasks, creating a potential for distributed deployment, and thus making ASC2 highly scalable.

Modularity The importance of the agent's cognitive capabilities like preferences, norm reasoning, goals and desires was already discussed, however, each of these aspects is studied in distinct research communities resulting in different, valid and interesting theories. From the experience of this dissertation, especially from a research standpoint, it is desirable for a framework to be able to easily embed these theories within the agents to allow for in-depth, comparative experimentation. ASC2 agent are actor micro-systems, consisting of multiple components that can communicate through internal messaging. This makes it quite easy to modify the framework by simply replacing a component. This is illustrated for example in Chapter 5 where normative advisor agents are created by replacing their logic reasoning engine with a norm reasoning framework.

> **Research Question 2:** *How can social agents utilize software and infrastructural models or entities?*

Software and infrastructural entities are one of the three main categories of the models needed for modelling a norm-governed socio-technical system alongside governing norms and participating actors. In the context of this dissertation, the focus is on allowing for an easy integration of the agent models with arbitrary external software. As it was discussed in Chapter 1, this is not a requirement classically recognized by the BDI literature, hence it is not satisfied by most existing BDI-frameworks. However, recent literature shows the interest in integrating agents into other software eco-systems, such as micro-services [46], web services and service oriented architecture [138], and traffic simulators [10], and even machine-learning software [115], and many of these works describe this integration as challenging. The experience from this dissertation reinforced this, the challenge of integrating software components with agents limits and demotes experimentation.

ASC2 was designed with interoperability as a first class requirement. The cross-compilation step makes sure that the agent models are compatible with any external software that is compatible with JVM-based programming languages, with minimal effort, which intuitively covers a vast majority of production-level software ecosystems. This turned out to be extremely useful, essentially being able to *hook* an agent or a whole MAS into any software setting means easy experimentation set-ups, which allows for the models themselves to be the main focus instead of trying to communicate with external software, as for example network or traffic simulators [1]. This is thoroughly covered in Chapters 2 and 4.

[1] Although never mentioned in the text, ASC2 agents already come with REST and gRPC interfaces out-of-the-box. For example, after running a MAS instance, there are HTTP/REST

Research Question 3: *How can social agents reason with and about norms?*

The concept of norms for the purposes of this dissertation covers a wide range of ideas in creating a normative multi-agent system. Norms can be personal policies of individual agents, contractual agreements that groups of agents can utilize to coordinate their collective goals, high-level societal laws that the actors can take into account in their decision-making, rules ascribed to (software) agents to make sure they behave within certain boundaries, or even a set of external rules, which do not affect the behavior of actors, but are used to monitor the behavior of the system as a whole. A normative multi-agent system then can include more than one of the aforementioned types of norms implemented in it. Furthermore, while the eFLINT norm reasoner has been predominantly utilized, this dissertation does not assume any specific type of norm reasoning, this allows for more flexibility and experimentation.

To cover all of these areas, Chapter 5 considers the capability for the agents to have an institutional perspective over their environment by introducing the concept of a norm instance: the institutional state of the environment, built upon a normative source and continuously updated via observations. Furthermore, Chapter 5 proposes a set of requirements for social agents based on [22] to allow them to *reason with and about norms*, agents should be:

- able to adopt or drop any number of norm sources as norm instances;
- able to qualify observations about their environment as normatively relevant updates, and conversely to respond to normative events by acting accordingly in their environment;
- able to query, update, revert and reset a normative state of any norm instance;
- able to receive and process or ignore normative events (e.g. new claims and liberties)
- able to follow or violate normative conclusions (e.g. obligations) or query responses (e.g. permissions and prohibitions)
- able to modify any of the above abilities at run-time.

---

end-points like `http://host/agent/achieve` that can be queried to communicate with each agent, and vice-versa, the agents can use Java's `java.net.HttpUrlConnection` package to communicate with other HTTP/REST end-points. As the agent's internal entities (goals, beliefs, preferences) are also Java objects, they can simply be (de-)serialized to/from JSON. These were used in multiple internal experiments, including network and traffic simulators.

Based on these requirements, the concept of normative advisors is introduced. Normative advisors are entities that enable the social agents to communicate with one or more external norm reasoners. Then, the tasks of maintaining an institutional perspective (state) and reasoning about specific sets of norms is delegated to the advisors while the social agent can maintain its autonomy regardless of what norm sources it has adopted. ASC2's modularity vastly helps in facilitating the implementation of this approach, but it is by no means exclusive to it, and because of the simplicity of the method almost any other BDI framework can be utilized in the same manner. Guidelines on how this can be done in other frameworks and what are the requirements, both for the BDI framework and the norm reasoner, are also discussed.

> **Research Question 4:** *How can we model desires and preferences of agents?*

To be soundly applied in support for policy-making agent models should manifest to some extent human-like behavior. For *traceability* and *explainability* reasons, decision-making concerning actions need to be analysed alongside the actions themselves (e.g. For which purpose the agent is asking access to the resource? On which basis the infrastructure is granting access?). Furthermore, modelling agents to manifest traceable and explainable decision-making whilst requiring them to reason with norms is a challenging task. It is very easy to see that an agent's decisions can be in conflict with a set of societal norms, two sets of norms adopted by the agent can be in conflict with each-other, or even the agent's desires or goals alone can result in non-trivial conflicts. To address these issues, this dissertation proposes giving the agent designer the ability to encode conditional and context-based higher level decision-making rules in agent programs in the form of explicit preferences.

Preferences play a crucial role in decision-making [135]. Several models of preferences have been presented in the literature (e.g. on decision-making, planning, etc.), with various levels of granularity and expressiveness [102, 30]. On a higher level, preference representation methods can be divided into *quantitative* and *qualitative* [102]. The most straightforward *quantitative* approaches are based upon *utility theory* and related forms of decision theory. In [48] one can find some examples of integration of these types of preferences in a BDI architecture.

Although quantitative approaches have clear computational advantages, they also suffer from the non-trivial issue of elicitation: translating users' preferences into utility functions. This explains the existence of a family of *qualitative* or hybrid solutions, as LPP [16] and PDDL3 [83]. Some preference models, as CP-nets (qualitative) [29] and GAI networks (quantitative) [88], have been specifically introduced for taking into account dependencies and conditions between preferences via *compact representations* [135], highly relevant in domains with a large number of features.

The strong support in the decision-making literature for *compact representations* of verbalized preferences—as for instance those captured e.g. by CP-nets [29]—motivates their use in computational agents, especially in applications in which agents are deemed to reproduce human behaviour, aiming to capture intentional characterizations of (computational) behaviour of computational agents in data-sharing infrastructures in support of policy-making and regulation activities.

Chapter 3 introduces an approach for integrating CP-Net preferences into BDI agents at the language level, specifically utilized in abstract goal refinement. This allows the agents to take an abstract goal and concretize it in an incremental manner through a decision-making process based on conditional and context-based preferences explicitly defined by the designer. These conditions and contexts can include observations over the environment, agent's beliefs, adopted norms –the institutional view over the environment–, and external events.

Introducing explicit preferences in BDI scripts brings three advantages: (1) It increases the *representational depth*, capturing what is the rationale behind the priority in plan selection; (2) It makes agent models more readable and *explainable*, as choices are in principle transparently derived from the preferential structure; (3) It makes the programs more *reusable*: it is plausible that agents (e.g. representatives of organizations) in a certain domain might share the same procedural knowledge even when having different preferences, as much as that agents might change their policy without changing their procedural knowledge.

> **Research Question 5:** *How to make agent-based modelling a usable approach for policy-makers and designers?*

Another issue that is addressed in this dissertation is the practicality of utilizing agent-based models as part of real-world system design as part of policy-making processes. It is always the case that accessibility and usability of the tools in a certain methodology is an important part of their adoption, it is hard and often even infeasible to convince domain experts like programmers to utilize an approach if they need to also adopt a whole new set of tools and ecosystems. This is the case for utilizing agent-based models and has been a major concern in this work, which has focused in particular on the model designer role in the policy-making process.

The advances in software development tools in recent years is intuitively both a symptom and a cause of the advancement of the computer software industry. This includes integrated development environments, build tools, automated testing frameworks, DevOps systems, automated deployment tools, code analysis, and many others support instruments. This dissertation argued that having these tools also available for agent-based programming brings a huge advantage to the field. Chapter 4 focuses on this idea, and discusses what are the design requirements of an agent-based programming framework that would make viable interfacing with these tools, and what would be the advantages for agent-based modelling. These

advantages include for example straightforward automated agent unit testing, automated multi-agent scenario execution, recording historical model executions traces, and a modular and collaborative system modelling that scales up the development process. The proof of concept show-cased in ASC2 is also illustrated in Chapter 4 with a focus on agent and multi-agent system testing.

8.4 Discussion, Limitations and Trade-offs

In this section, the main limitations and challenges of the work presented in this dissertation will be discussed. Furthermore, important trade-offs that occurred by the choices taken are analyzed to provide the reader with better insights about the side-effects of these choices.

Theoretical depth vs. Scalability Throughout this thesis, scalability, both in model development and execution, is a central theme. Ideally, this should not take anything away from the underlying theoretical soundness of the work; however, in practice there is always a trade-off. A good example of this can be a comparison between ASC2 and other frameworks like 2APL [54] and Jason [26] which both are sources of inspiration for ASC2.

2APL agents on the language-level have declarative knowledge about the (expected) effects of their action, known as belief-update actions. In effect, this is a type of introspection ability, which, although interesting for theoretical purposes, has a massive impact on performance: the agent is not only executing actions, but also monitoring its internal knowledge about the known effects of its actions while performing them. In fact, 2APL was taken out from the benchmarks of Chapter 2 as it could not complete them in reasonable time and would be an outlier in the results. The choice for ASC2's design to go the opposite way was not taken lightly, though. We explored the advantages of having such constructs [123] (paper not included in the thesis), where belief-update actions (expected effects of primitive actions) are added to ASC2 and utilized for preference reasoning in the plan selection step of the agent. While this reasoning happens at compile time and does not affect the reactive nature of the agent, in practice assuming knowledge about effects of actions is unrealistic in a dynamic environment.

Another example is the introspection of agents into its own procedural knowledge – its own plan library – which can allow the agent to *plan* ahead before selecting a plan, by analysing the possible outcomes of that plan. There are multiple approaches introduced in the literature to allow agents to reason about through planning based on their procedural knowledege (see [151, 161]. We explored a similar idea in another work [122] (not included in this thesis), to use run-time introspection and preference reasoning for plan selection of the agent. The problem with this approach is that it assumes that the agent can reason about multiple steps into its goal refinement. Intuitively, this is not feasible in

highly dynamic environments [57] and becomes a computational burden. These reasons, together with space constraints, explain why these two papers [122, 123] are not presented in this dissertation.

As a final example of a trade-off, we can look at the three selection functions of AgentSpeak(L)/Jason and their counterpart in ASC2: goal-selection, plan selection, and intention selection. There are important theoretical reasons for the presence of these selection functions (see [140, 139]), for example the goal selection function can be utilized by the agent to select the most important or most preferred goal for refinement at run-time. However, in ASC2, in practice, only plan selection exists. The reason for this is the full asynchronous nature of the agents. Unlike Jason and many other frameworks, ASC2 does not include a real execution cycle starting from goal selection and ending in intention selection. Instead, the agents are fully concurrent and reactive entities, a reasoning process starts either when there is an event and the agent has computational resources available (CPU), or alternatively when computational resources become available and the agent has some events it did not react to yet, and the only way for the agent to perform goal selection for example is to use a priority queue to sort the incoming events with any arbitrary order. Nevertheless, the effect of having concurrent agents is illustrated in Section 2.4.3 where ASC2 has a massive performance advantage –or better put, absence of thread blocking between intentions– over Jason.

Testability vs. Verifiability In system and software verification, in comparison to formal methods such as static analysis, theorem proving, and model checking, software testing is considered to be a less exhaustive approach to verify the correct behavior of a system. Multi-agent systems are also not an exception, there are multiple approaches introduced in the literature on verification of agent systems (for a comprehensive overview see [79]). Indeed, this can be considered a limitation of this dissertation, although the soundness of some parts of the introduced policy-making framework are proven (e.g., correctness of basic CP-Net preference reasoning in Chapter 3), testing is still the main approach to verification, as some other parts of the framework that although illustrated to be correct through examples, are not fully verified in a formal way (e.g., context-based CP-Net preferences in Chapter 3).

Although testing in itself has many advantages over formal methods (testing are generally performed on the real system and not on a model), this limitation must be acknowledged. However, we should also look at the context of the agent systems in this dissertation, and why testing was favored over formal methods. In our policy-making framework, agents and the multi-agent systems are utilized as models of a real system under scrutiny, its participating actors and implemented policies for the sake of verifying the said system and policies. In a sense, the agents are already a model of a real system, so creating further second-level models of the agents for the sake of model checking for example and proving their correctness,

although theoretically interesting and valuable in other contexts, could be an impractical exercise. Nevertheless, we still believe that the framework as a whole can benefit from formal methods and on-going research is being performed for example to allow model checking a norm source.

Nondeterministic behavior Chapter 1 argued that agent-based modelling is a suitable approach for policy-making because it can create a system with emergent behavior from lower-level actor specifications. This is true with ASC2 agents –or any BDI-like agent– because they react and behave based on their perception of their environment. However, ASC2 agents are fully deterministic in their beliefs and decision-making; this is desirable for cases where individual agent behavior is analyzed and reproducibility is a requirement. However, for larger models where the individual agent behavior is not as important as the statistical emergent behaviors of the system, allowing for probabilistic beliefs and decision-making is an important asset. The reason for this is that instead of modelling many different types of deterministic agents, the designer can create agents with probabilistic decision-making and then generate a synthetic population of agents with different attributes which results in different behaviors.

Note that ASC2 already implicitly allows for such agents. One approach to achieve this is utilizing the context-conditions of plans, where a plan's applicability is affected by some arbitrary nondeterministic argument. Take for example the following plan:

```
+!g : c & #Random.nextDouble() > 0.5 => ...
```

This plan is essentially applicable if its context condition holds and an arbitrary random number selection between zero and one is above 0.5, meaning it is not selected 50% of the times, even if it is applicable. ASC2 even allows for generation of a population of agents through the same approach, take for example the following agent:

```
change_of_decision(#Random.nextDouble()).
+!g : c & change_of_decision(P) & P > 0.5 => ...
```

With this script, at initialization time, each individual agent calls the underlying `#Random.nextDouble()` and puts a different variable as its belief. Then, at run-time and when deliberating about the applicability of the plan, checks if this variable is above a certain threshold. Note that this variable can even be subject to changes at run-time. Finally, the most interesting and unique way for ASC2 agents to model probabilistic behavior is through preferences. Recall the following preference statement from Chapter 3:

```
!go_order(L,_) >> !go_order(_,_) :- at(L) .
```

This statement essentially is equal to "I prefer the place I am already at (over all others)", we can change this statement to:

```
!go_order(L,_) >> !go_order(_,_) :- at(L) & #Random.nextDouble() < 0.8.
```

Which makes it equal to the statement that "I prefer the place I am already at with 80% probability", making it a nondeterministic preference.

Although ASC2 in its current form gives provides these capabilities for encoding probabilistic behavior, it needs a wider array of study and modifications to allow for more expressive models that include nondeterministic properties, both concerning the agent and the environment.

8.5 Future Works

This section shortly explores some of the ideas that were considered during this research but never fully executed as possible future works.

8.5.1 Macro System Modelling

As mentioned as a limitation, approaches in this dissertation include only agents with deterministic beliefs, rules, and plans exclusively modeled by a human designer. This is an advantage for the cases where the goal is to model a society bottom-up and analyse the behaviors of a handful of agents. The main drawback of this method is that it limits the size of the models in terms of diversity of agent types; although the framework allows for a large population of agents from the computational perspective, with the current capabilities of ASC2, the designer may not be able to model this population in a sufficiently realistic manner. Then, approaches for top-down modelling of the system by generating a synthetic population are desirable, specially if such approach still ends up in expressive agent scripts that can be analyzed.

Generating synthetic populations is a mature field of study[2] and there are multiple algorithms and methods proposed for generation of agent populations [39, 101], some even with capabilities for generating executable agent-based models [84]. These approaches are typically categorized based on their input data, namely into sample-based and sample-free methods, or based on their synthesis technique into synthetic reconstruction (SR) and combinatorial optimization (CO) methods (see [38]). Another thread of work that is specifically interesting for integration with ASC2 are methods for generating statistical cultures (or random elections) [155] that are effectively algorithms for generating a synthetic population represented by their preferences in a certain domain, with a focus on realism and diversity of the population. Intuitively, this fulfills one of the main motivations of integrating preferences into agents which is to have agents with shared procedural knowledge behave differently based on their preference. This allows the designer to model

[2]See for example the *Journal of Artificial Societies and Social Simulation*: `https://www.jasss.org/JASSS.html`

detailed agent scripts and preferences and then generate a large population of agents based on statistical cultures. Figure 8.1 is a simple illustration of using this approach.

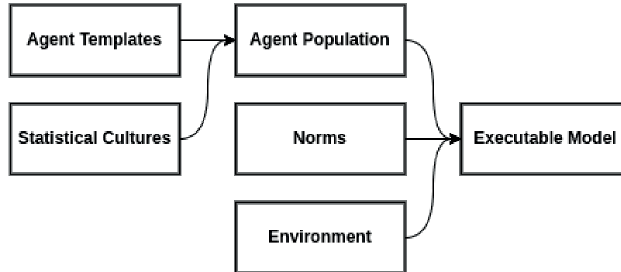

Figure 8.1: Overview of modelling process with synthetic populations

8.5.2 Alternative Agent Reasoning Approaches

The next intuitive future work for this dissertation is the exploration of extending the agents' reasoning capabilities. This is especially interesting because of the modularity of ASC2's agent architecture: many extensions to BDI reasoning proposed in the literature that may be infeasible to implement in other framework should in principle be easier to utilize in ASC2.

Value-based reasoning Value-based reasoning in BDI agents have had increasing attention from the respective research community in recent years, particularly in situations that the agents are required to make moral or ethical decisions. Values are an important part of human reasoning [177] and value-based reasoning refers to the idea that agents should make their decisions based on a consistent and stable underlying value system that they desire to satisfy. There are works in the literature on embedding value-based reasoning BDI agents that perform a diverse set of tasks and utilize values as an over-arching mechanism of balance between these tasks (e.g., see [48]). Values are also important in agents with norm reasoning, as it was also argued in Chapter 5 norms are rarely absolute and different set of norms often are in conflict, meaning conflict resolution methods are required. Values can be a basis for conflict resolution and are used as such in the literature [12], between the norms of "user's privacy must be preserved" and "user's health must be preserved" which one should take precedence when there is a conflict? This intuitively ties in strongly to this dissertation as the idea of normative conflict resolution is an important part of it and furthermore answering questions like what is the connection between preferences and values.

Planning Planning from first principles in BDI agents is an idea that has been explored with various levels of success in the literature [151, 58, 142]. Among different planning methods, HTN-like approaches [75] are particularly interesting for BDI agents, as they are both based on decomposition of tasks into smaller tasks. While interesting to consider, (deliberate) planning in BDI agents is not always an enhancement for multiple reasons. Firstly, planning requires introspection: the agent needs to look into its own internal procedural knowledge to be able to create plans, this means a considerable performance impact and less scalability. Secondly, concretized plans tend to fail unexpected changes in the environment, meaning planning is less feasible in highly dynamic environments [57].

However, as illustrated in [122, 123] on-demand planning prior to action at run-time is not the only method for an agent to utilize planning. As integration of norms in agent's reasoning has an impact on its decision-making, a future work can be to utilize planning to optimize agent's procedural knowledge with a bounded usage of computational power at compile- or run-time by taking norms as (soft) constraints in planning algorithm's domain information. Another interesting future work is to utilize planning for multi-agent coordination; planning algorithms are typically not constrained to creating plans for only one agent, and this capability is lost when they are used from the perspective of only one agent. This can allow agents to create high-level shared plans to allow them to coordinate in achieving their tasks.

(Sub-symbolic) machine learning Interactions between machine learning capabilities and BDI agents are another idea that have been explored in the literature [153, 90]. These studies generally revolve around using simple machine learning algorithms (e.g., Q-Learning) to enhance the decision-making of the agent. The issue here is that with recent advancement in AI algorithms, from the performance and behavioral complexity perspective, symbolic approaches such as BDI agents that require a designer to model the agents are not on-par with sub-symbolic methods that can infer behavior based on patterns in large amounts of data. This means if the goal is to have more realistic behavior, the BDI component is just a burden for the agent.

However, sub-symbolic approaches also have shortcomings, they struggle with explainability and interpretability [111] and there is a pressing research interest to amend this in communities like Explainable Artificial Intelligence (XAI) [52]. Furthermore, sub-symbolic algorithms are very challenging to regulate and verify from high level perspective like normativity or fairness [110]. We believe there are opportunities to exploit in utilizing symbolic approaches and frameworks like what is presented in this work to address these issues in the future. As an example, in the architecture of normative advisors, one of the motivations of separating the normative reasoning from agent reasoning is to isolate the two so that, in theory, any agent, including sub-symbolic ones, can also utilize advisors via the same

interface to create normative sub-symbolic agents. This way, not only the agent can query about normativity of its actions prior to performance, the advisor can also prompt the agent based on its internal institutional state to perform certain actions. This results in a separation of the agent's reasoning that it has learned via processing large amounts of data and effective norms that are encoded by experts.

Bibliography

[1] Software testing services market by product, end-users, and geography - global forecast and analysis 2019-2023. https://www.industryresearch.co/software-testing-services-market-14620379, Aug 2019. Online; accessed 1 January 2020.

[2] Additional Protocol I. Protocol Additional to the Geneva Conventions of 12 August 1949, and relating to the Protection of Victims of International Armed Conflicts (adopted 8 June 1977, entered into force 7 December 1978) 1125 UNTS 3, 1977.

[3] Tobias Ahlbrecht, Jürgen Dix, and Niklas Fiekas. Scalable multi-agent simulation based on mapreduce. In *Multi-Agent Systems and Agreement Technologies*, pages 364–371. Springer International Publishing, 06 2017.

[4] Rania Rizki Arinta and Emanuel Andi W.R. Natural disaster application on big data and machine learning: A review. In *2019 4th International Conference on Information Technology, Information Systems and Electrical Engineering (ICITISEE)*, pages 249–254, 2019.

[5] Malte Aschermann, Sophie Dennisen, Philipp Kraus, and Jörg P. Müller. LightJason, a Highly Scalable and Concurrent Agent Framework: Overview and Application. In *Proceedings of the 17th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS2018)*, pages 1794–1796, 2018.

[6] Malte Aschermann, Philipp Kraus, and Jörg P. Müller. Lightjason - a bdi framework inspired by jason. In *EUMAS/AT*, 2016.

[7] Jorge A Baier and Sheila A McIlraith. On domain-independent heuristics for planning with qualitative preferences. In *AAAI Spring Symposium: Logical Formalizations of Commonsense Reasoning*, pages 7–12, 2007.

[8] Najwa Abu Bakar and Ali Selamat. Agent systems verification : systematic literature review and mapping. *Applied Intelligence*, 48(5):1251–1274, 2018.

[9] Matteo Baldoni, Cristina Baroglio, Katherine M. May, Roberto Micalizio, and Stefano Tedeschi. Computational accountability in mas organizations with adopt. *Applied Sciences (Switzerland)*, 8, 3 2018.

[10] Jetze Baumfalk, Mehdi Dastani, Barend Poot, and Bas Testerink. A sumo extension for norm-based traffic control systems. In *Simulating Urban Traffic Scenarios: 3rd SUMO Conference 2015 Berlin, Germany*, pages 55–82. Springer, 2019.

[11] Tristan M. Behrens and Jürgen Dix. Model checking multi-agent systems with logic based Petri nets. *Annals of Mathematics and Artificial Intelligence*, 51(2-4):81–121, 2007.

[12] Trevor Bench-Capon and Sanjay Modgil. Norms and value based reasoning: justifying compliance and violation. *Artificial Intelligence and Law*, 25:29–64, 2017.

[13] Trevor Bench-Capon, Henry Prakken, Adam Wyner, and Katie Atkinson. Argument schemes for reasoning with legal cases using values. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Law*, ICAIL '13, pages 13–22, New York, NY, USA, 2013. ACM.

[14] Trevor J. M. Bench-Capon. Persuasion in Practical Argument Using Value-based Argumentation Frameworks. *Journal of Logic and Computation*, 13(3):429–448, 06 2003.

[15] Trevor J. M. Bench-Capon and Katie Atkinson. Dimensions and values for legal CBR. In Adam Z. Wyner and Giovanni Casini, editors, *Legal Knowledge and Information Systems - JURIX 2017: The Thirtieth Annual Conference, Luxembourg, 13-15 December 2017*, volume 302 of *Frontiers in Artificial Intelligence and Applications*, pages 27–32. IOS Press, 2017.

[16] Meghyn Bienvenu, Christian Fritz, and Sheila A McIlraith. Planning with qualitative temporal preferences. *KR*, 6:134–144, 2006.

[17] Sebastian Blessing, Kiko Fernandez-Reyes, Albert Mingkun Yang, Sophia Drossopoulou, and Tobias Wrigstad. Run, actor, run: Towards cross-actor language benchmarking. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE 2019, page 41–50, New York, NY, USA, 2019. Association for Computing Machinery.

[18] Guido Boella, Llio Humphreys, Robert Muthuri, Piercarlo Rossi, and Leendert van der Torre. A critical analysis of legal requirements engineering from the perspective of legal practice. *2014 IEEE 7th International Workshop on Requirements Engineering and Law, RELAW 2014 - Proceedings*, pages 14–21, 2014.

[19] Guido Boella, Llio Humphreys, Robert Muthuri, Piercarlo Rossi, and Leendert van der Torre. A critical analysis of legal requirements engineering from the perspective of legal practice. *2014 IEEE 7th International Workshop on Requirements Engineering and Law, RELAW 2014 - Proceedings*, pages 14–21, 2014.

[20] Guido Boella, Gabriella Pigozzi, and Leendert van der Torre. Normative systems in computer science-ten guidelines for normative multiagent systems. *Normative Multi-Agent Systems*, pages 1–21, 2009.

[21] Guido Boella and Leendert van der Torre. Regulative and constitutive norms in normative multiagent systems. In *Proceedings of the Ninth International Conference on Principles of Knowledge Representation and Reasoning*, KR'04, page 255–265. AAAI Press, 2004.

[22] Guido Boella, Leendert van der Torre, and Harko Verhagen. Introduction to normative multiagent systems. *Computational and Mathematical Organization Theory*, 12(2-3 SPEC. ISS.):71–79, 2006.

[23] William H. Boothby. *New Technologies and the Law of War and Peace*. Cambridge University Press, Cambridge, 2019.

[24] Rafael H. Bordini, Michael Fisher, Willem Visser, and Michael Wooldridge. Verifiable multi-agent programs. *Lecture Notes in Artificial Intelligence (Subseries of Lecture Notes in Computer Science)*, 3067:72–89, 2004.

[25] Rafael H. Bordini, Michael Fisher, Willem Visser, and Michael Wooldridge. Verifying multi-agent programs by model checking. *Autonomous Agents and Multi-Agent Systems*, 12(2):239–256, 2006.

[26] Rafael H Bordini, Jomi F Hübner, and Renata Vieira. Jason and the golden fleece of agent-oriented programming. *Multi-agent programming: languages, platforms and applications*, pages 3–37, 2005.

[27] Juan A. Botía, Jorge J. Gómez-Sanz, and Juan Pavón. Intelligent data analysis for the verification of multi-agent systems interactions. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4224 LNCS:1207–1214, 2006.

[28] Paul-Marie Boulanger and Thierry Bréchet. Models for policy-making in sustainable development: The state of the art and perspectives for research. *Ecological Economics*, 55:337–350, 11 2005.

[29] Craig Boutilier. CP-nets: A Tool for Representing and Reasoning with Conditional. *Ai Applications*, 21:135–191, 2004.

[30] Ronen Brafman and Carmel Domshlak. Preference Handling - An Introductory Tutorial. *AI Magazine*, 30(1):58, 2009.

[31] Michael Bratman. Intention, plans, and practical reason. 1987.

[32] G Brewka. A Rank Based Description Language for Qualitative Preferences. *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI'04)*, (May):303–307, 2004.

[33] Jan Broersen, Mehdi Dastani, Joris Hulstijn, Zisheng Huang, and Leendert van der Torre. The BOID Architecture - Conflicts Between Beliefs, Obligations, Intentions and Desires. In *In Proceedings of the Fifth International Conference on Autonomous Agents*, pages 9–16. ACM Press, 2001.

[34] Jan Broersen, Mehdi Dastani, and Leendert van der Torre. Resolving conflicts between beliefs, obligations, intentions, and desires. *Lecture Notes in Artificial Intelligence (Subseries of Lecture Notes in Computer Science)*, 2143:568–579, 2001.

[35] Rafael C Cardoso, Angelo Ferrando, Louise A Dennis, and Michael Fisher. Implementing Ethical Governors in BDI. In Natasha Alechina, Matteo Baldoni, and Brian Logan, editors, *Engineering Multi-Agent Systems*, pages 22–41, Cham, 2022. Springer International Publishing.

[36] Rafael C. Cardoso, Maicon R. Zatelli, Jomi F. Hübner, and Rafael H. Bordini. Towards benchmarking actor- and agent-based programming languages. In *Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! 2013, page 115–126, New York, NY, USA, 2013. Association for Computing Machinery.

[37] Vinay Chamola, Vikas Hassija, Sakshi Gupta, Adit Goyal, Mohsen Guizani, and Biplab Sikdar. Disaster and pandemic management using machine learning: A survey. *IEEE Internet of Things Journal*, 8(21):16047–16071, 2021.

[38] Kevin Chapuis, Patrick Taillandier, and Alexis Drogoul. Generation of synthetic populations in social simulations: A review of methods and practices. *Journal of Artificial Societies and Social Simulation*, 25(2):6, 2022.

[39] Kevin Chapuis, Patrick Taillandier, Benoit Gaudou, Frédéric Amblard, and Samuel Thiriot. Gen*: an integrated tool for realistic agent population synthesis. In *Conference of the European Social Simulation Association*, pages 189–200. Springer, 2019.

[40] Dominik Charousset, Raphael Hiesgen, and Thomas C. Schmidt. Caf - the c++ actor framework for scalable and resource-efficient applications. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*, AGERE! '14, page 15–28, New York, NY, USA, 2014. Association for Computing Machinery.

[41] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 2000.

[42] Sylvan Clebsch and Sophia Drossopoulou. Fully concurrent garbage collection of actors on many-core machines. *SIGPLAN Not.*, 48(10):553–570, October 2013.

[43] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! 2015, page 1–12, New York, NY, USA, 2015. Association for Computing Machinery.

[44] Chris Clifton, Murat Kantarcioğlu, AnHai Doan, Gunther Schadow, Jaideep Vaidya, Ahmed Elmagarmid, and Dan Suciu. Privacy-preserving data integration and sharing. In *Proceedings of the 9th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery*, pages 19–26, 2004.

[45] Roberta Coelho, Uirá Kulesza, Arndt von Staa, and Carlos Lucena. Unit testing in multi-agent systems using mock agents and aspects. In *International Workshop on Software Engineering for Large-Scale Multi-Agent Systems*, SELMAS '06, page 83–90, 2006.

[46] R.W. Collier, D. Lillis, E. O'Neill, and G.M.P. O'Hare. MAMS: Multi-agent microservices. *The Web Conference 2019 - Companion of the World Wide Web Conference, WWW 2019*, 2019.

[47] Geoffrey S. Corn. War, law, and the oft overlooked value of process as a precautionary measure. *Pepperdine Law Review*, 42:419–466, 2014.

[48] Stephen Cranefield, Michael Winikoff, Virginia Dignum, and Frank Dignum. No pizza for you: Value-based plan selection in BDI agents. *IJCAI International Joint Conference on Artificial Intelligence*, pages 178–184, 2017.

[49] N. Criado, E. Argente, P. Noriega, and V. Botti. Towards a normative BDI architecture for norm compliance. *CEUR Workshop Proceedings*, 627:65–81, 2010.

[50] Natalia Criado, Estefania Argente, Pablo Noriega, and Vicente Botti. Manea: A distributed architecture for enforcing norms in open mas. *Engineering Applications of Artificial Intelligence*, 26(1):76–95, 2013.

[51] Rebecca Crootof. The Killer Robots are here: Legal and Policy Implications. *Cardozo Law Review*, 36:1837–1915, 2015.

[52] Arun Das and Paul Rad. Opportunities and challenges in explainable artificial intelligence (xai): A survey. *arXiv preprint arXiv:2006.11371*, 2020.

[53] Aniruddha Dasgupta and Aditya K. Ghose. Implementing reactive BDI agents with user-given constraints and objectives. *International Journal of Agent-Oriented Software Engineering*, 4(2):141, 2010.

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

[55] Mehdi Dastani, Jürgen Dix, and Peter Novak. The first contest on multi-agent systems based on computational logic. In *Proceedings of the 6th International Conference on Computational Logic in Multi-Agent Systems*, CLIMA'05, page 373–384, Berlin, Heidelberg, 2005. Springer-Verlag.

[56] Nuno David, Jaime Simão Sichman, and Helder Coelho. Towards an emergence-driven software process for agent-based simulation. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2581(301041):89–104, 2003.

[57] Lavindra de Silva and Lin Padgham. A Comparison of BDI Based Real-Time Reasoning and HTN Based Planning. In *AI 2004: Advances in Artificial Intelligence*, pages 1167–1173, 2004.

[58] Lavindra de Silva, Lin Padgham, and Sebastian Sardina. HTN-like solutions for classical planning problems: An application to BDI agent systems. *Theoretical Computer Science*, 763:12–37, 2019.

[59] Defense Innovation Board. AI Principles : Recommendations on the Ethical Use of Artificial Intelligence by the Department of Defense Defense Innovation Board. Technical report, Department of Defense, 2019.

[60] Ameneh Deljoo, Tom M van Engers, Robert van Doesburg, Leon Gommans, and Cees de Laat. A normative agent-based model for sharing data in secure trustworthy digital market places. In *Proceedings of the 10th International Conference on Agents and Artificial Intelligence*, pages 290–296, 2018.

[61] Yuri Demchenko, Paola Grosso, Cees de Laat, and Peter Membrey. Addressing big data issues in scientific data infrastructure. In *2013 International Conference on Collaboration Technologies and Systems (CTS)*, pages 48–55, 2013.

[62] Louise A. Dennis, Michael Fisher, Nicholas K. Lincoln, Alexei Lisitsa, and Sandor M. Veres. Practical verification of decision-making in agent-based autonomous systems. *Automated Software Engineering*, 23(3):305–359, 2016.

[63] Louise A. Dennis, Michael Fisher, Matthew P. Webster, and Rafael H. Bordini. Model checking agent programming languages. *Automated Software Engineering*, 19(1):5–63, 2012.

[64] Enrico Denti, Andrea Omicini, and Alessandro Ricci. Tu prolog: A lightweight prolog for internet applications and infrastructures. In *Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages*, PADL '01, page 184–198, Berlin, Heidelberg, 2001. Springer-Verlag.

[65] Akshat Dhaon and Rem W. Collier. Multiple inheritance in agentspeak(l)-style programming languages. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*, New York, NY, USA, 2014. Association for Computing Machinery.

[66] Tommaso Di Noia, Thomas Lukasiewicz, Maria Vanina Martinez, Gerardo I. Simari, and Oana Tifrea-Marciuska. Combining existential rules with the power of CP-theories. *IJCAI International Joint Conference on Artificial Intelligence*, 2015-Janua(Ijcai):2918–2925, 2015.

[67] Frank Dignum, Virginia Dignum, and Catholijn Jonker. Towards agents for policy making. pages 141–153, 05 2008.

[68] Frank Dignum, David Kinny, and Liz Sonenberg. Motivational attitudes of agents: On desires, obligations, and norms. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2296(Section 2):83, 2002.

[69] Jürgen Dix and Yingqian Zhang. *Impact: A Multi-Agent Framework with Declarative Semantics*, volume 15, pages 69–94. 01 2005.

[70] Carmel Domshlak, Eyke Hüllermeier, Souhila Kaci, and Henri Prade. Preferences in AI: An overview. *Artificial Intelligence*, 175(7-8):1037–1052, 2011.

[71] Giovanni Dosi and Andrea Roventini. More is different ... and complex!: The case for agent-based macroeconomics. *SSRN Electronic Journal*, 01 2019.

[72] Paul Ducheine and Terry Gill. From Cyber Operations to Effects: Some Targeting Issues. *Militair Rechtelijk Tijdschrift*, 111(3):37–41, 2018.

[73] Erdem Eser Ekinci, Ali Murat Tiryaki, Övünç Çetin, and Oguz Dikenelli. Goal-Oriented Agent Testing Revisited. In Michael Luck and Jorge J Gomez-Sanz, editors, *Agent-Oriented Software Engineering IX*, pages 173–186, 2009.

[74] A.F.S.A. El Gammal. *Towards a comprehensive framework for business process compliance*. Other publications tisem, Tilburg University, 2012.

[75] Kutluhan Erol, James Hendler, and Dana S. Nau. HTN planning: Complexity and expressivity. *In Proceedings of the 12th AAAI Conference on Artificial Intelligence*, pages 1123–1129, 1994.

[76] Jacques Ferber and Gerhard Weiss. *Multi-agent systems: an introduction to distributed artificial intelligence*, volume 1. Addison-wesley Reading, 1999.

[77] Angelo Ferrando, Louise A. Dennis, Davide Ancona, Michael Fisher, and Viviana Mascardi. *Verifying and validating autonomous systems: Towards an integrated approach*, volume 11237. Springer International Publishing, 2019.

[78] Michael Fisher, Rafael H. Bordini, Benjamin Hirsch, and Paolo Torroni. Computational logics and agents: A road map of current technologies and future trends. *Computational Intelligence*, 23(1):61–91, 2007.

[79] Michael Fisher, Viviana Mascardi, Kristin Yvonne Rozier, Bernd Holger Schlingloff, Michael Winikoff, and Neil Yorke-Smith. *Towards a framework for certification of reliable autonomous systems*, volume 2. Springer US, 2020.

[80] Dieter Fleck, editor. *The Handbook of International Humanitarian Law*. Oxford University Press, Oxford, 3 edition, 2013.

[81] Peter Fratrič, Mostafa Mohajeri Parizi, Giovanni Sileno, Tom van Engers, and Sander Klous. Do agents dream of abiding by the rules? learning norms via behavioral exploration and sparse human supervision. *Nineteenth International Conference on Artificial Intelligence and Law (ICAIL 2023), 2023, Braga, Portugal*, 1.

[82] Peter Fratrič, Giovanni Sileno, Sander Klous, and Tom van Engers. Manipulation of the bitcoin market: an agent-based study. *Financial Innovation*, 8(1):60, 2022.

[83] Alfonso Gerevini and Derek Long. Plan Constraints and Preferences in PDDL3 The Language of the Fifth International Planning Competition 1 Motivations and Goals. pages 1–12, 2005.

[84] Amineh Ghorbani, Pieter Bots, Virginia Dignum, and Gerard Dijkema. MAIA: A framework for developing agent-based social simulations. *Journal of Artificial Societies and Social Simulation*, 16(2), 2013.

[85] Jack P Gibbs. Norms: The problem of definition and classification. *American Journal of Sociology*, 70(5):586–594, 1965.

[86] Leon Gommans. *Multi-Domain Authorization for e-Infrastructures*. PhD thesis, 2014.

[87] Leon Gommans, John Vollbrecht, Betty Gommans de Bruijn, and Cees de Laat. The service provider group framework. *Future Generation Computer Systems*, 45:176–192, 2014.

[88] C. Gonzales and P. Perny. GAI Networks for Utility Elicitation. *Proceedings of the 9th International Conference on the Principles of Knowledge Representation and Reasoning*, pages 224–233, 2004.

[89] Matthias Grabmair. Predicting trade secret case outcomes using argument schemes and learned quantitative value effect tradeoffs. In *Proceedings of the 16th edition of the International Conference on Articial Intelligence and Law*, page 89–98, New York, NY, USA, 2017. Association for Computing Machinery.

[90] Alejandro Guerra-Hernández, Amal El Fallah-Seghrouchni, and Henry Soldano. Learning in bdi multi-agent systems. In *Computational Logic in Multi-Agent Systems: 4th International Workshop, CLIMA IV, Fort Lauderdale, FL, USA, January 6-7, 2004, Revised Selected and Invited Papers 4*, pages 218–233. Springer, 2005.

[91] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2):202 – 220, 2009. Distributed Computing Techniques.

[92] Herbert Lionel Adolphus Hart, Herbert Lionel Adolphus Hart, Joseph Raz, and Leslie Green. *The concept of law*. oxford university press, 2012.

[93] Md Morshadul Hasan, József Popp, and Judit Oláh. Current landscape and influence of big data on finance. *Journal of Big Data*, 7, 12 2020.

[94] Mustafa Hashmi and Guido Governatori. Norms modeling constructs of business process compliance management frameworks: a conceptual evaluation. *Artificial Intelligence and Law*, 26:251–305, 9 2018.

[95] Mustafa Hashmi, Guido Governatori, Ho Pun Lam, and Moe Thandar Wynn. Are we done with business process compliance: state of the art and challenges ahead. *Knowledge and Information Systems*, 57:79–133, 10 2018.

[96] Andreas Herzig, Emiliano Lorini, Laurent Perrussel, and Zhanhao Xiao. Bdi logics for bdi architectures: Old problems, new perspectives. *KI - Künstliche Intelligenz*, 31(1):73–83, Mar 2017.

[97] Carl Hewitt. Actor model of computation: Scalable robust information systems, 2010.

[98] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, page 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[99] Koen V. Hindriks. Programming Rational Agents in GOAL. In *Multi-agent programming: Languages, platforms and applications*, chapter 4, pages 119–157. 2009.

[100] Wesley Newcomb Hohfeld. Fundamental legal conceptions as applied in judicial reasoning. *The Yale Law Journal*, 26(8):710–770, 1917.

[101] Nam Huynh, Johan Barthelemy, and Pascal Perez. A heuristic combinatorial optimisation approach to synthesising a population for agent-based modelling purposes. *Journal of Artificial Societies and Social Simulation*, 19(4):11, 2016.

[102] A Jorge and Sheila A. McIlraith. Planning with preferences. *AI Magazine*, 29(4):25–36, 2008.

[103] Quinta Jurecic. Paul C. Ney Jr., General Counsel, U.S. Department of Defense, Keynote Address at the Israel Defense Forces 3rd International Conference on the Law of Armed Conflict. *Lawfare*, may 2019.

[104] C. Kaiser and Jean-François Pradat-Peyre. Chameneos, a concurrency game for java, ada and others. In *ACS/IEEE International Conference on Computer Systems and Applications*, pages 62–70. IEEE, 08 2003.

[105] Timotheus Kampik and Juan Carlos Nieves. JS-son - A lean, extensible JavaScript agent programming library. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 12058 LNAI:215–234, 2020.

[106] Mohamed A. Khamis and Khaled Nagi. Designing multi-agent unit tests using systematic test design patterns (extended version). *Engineering Applications of Artificial Intelligence*, 26(9):2128–2142, 2013.

[107] Mohammed Khouj, César López, Sarbjit Sarkaria, and José Marti. Disaster management in real time simulation using machine learning. In *2011 24th Canadian Conference on Electrical and Computer Engineering(CCECE)*, pages 001507–001510, 2011.

[108] Vincent J. Koeman, Koen V. Hindriks, and Catholijn M. Jonker. Automating failure detection in cognitive agent programs. *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS*, pages 1237–1246, 2016.

[109] Jonathan Kwik, Tomasz Zurek, and Tom van Engers. Designing international humanitarian law into military autonomous devices. https://ssrn.com/abstract=4109286, May 2022.

[110] Derek Leben. Normative principles for evaluating fairness in machine learning. In *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*, AIES '20, page 86–92, New York, NY, USA, 2020. Association for Computing Machinery.

[111] Pantelis Linardatos, Vasilis Papastefanopoulos, and Sotiris Kotsiantis. Explainable ai: A review of machine learning interpretability methods. *Entropy*, 23(1):18, 2020.

[112] Andrea Loreggia, Emiliano Lorini, and Giovanni Sartor. A ceteris paribus deontic logic. In Francesco Calimeri, Simona Perri, and Ester Zumpano, editors, *Proceedings of the 35th Italian Conference on Computational Logic - CILC 2020, Rende, Italy, October 13-15, 2020*, volume 2710 of *CEUR Workshop Proceedings*, pages 248–262. CEUR-WS.org, 2020.

[113] Emiliano Lorini. Logics for games, emotions and institutions. *The IfCoLog Journal of Logics and their Applications*, 4(9):3075–3113, 2017.

[114] Michael Luck, Mark d'Inverno, et al. A normative framework for agent-based systems. *Computational & Mathematical Organization Theory*, 12(2):227–250, 2006.

[115] Marco Lützenberger, Sebastian Ahrndt, Nils Masuch, Axel Heßler, Benjamin Hirsch, and Sahin Albayrak. The bdi driver in a service city. In *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 3*, pages 1257–1258, 2011.

[116] Juliano Maranhão, Edelcio G. de Souza, and Giovanni Sartor. A dynamic model for balancing values. In *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Law*, ICAIL '21, page 89–98, New York, NY, USA, 2021. Association for Computing Machinery.

[117] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[118] D McDermott, M Ghallab, A Howe, C Knoblock, A Ram, M Veloso, D Weld, and D Wilkins. PDDL - The Planning Domain Definition Language. *The AIPS-98 Planning Competition Committee*, page 27, 1998.

[119] Felipe Meneguzzi and Lavindra De Silva. Planning in bdi agents: a survey of the integration of planning algorithms and agent reasoning. *The Knowledge Engineering Review*, 30(1):1–44, 2015.

[120] Felipe Meneguzzi and Michael Luck. Norm-based behaviour modification in bdi agents. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS*, volume 1, pages 177–184, 01 2009.

[121] Dejan Mitrovic, Mirjana Ivanovic, and Costin Badica. Jason agents in java ee environments. In *2013 17th International Conference on System Theory, Control and Computing (ICSTCC)*, pages 768–771, 10 2013.

[122] Mostafa Mohajeri Parizi, Giovanni Sileno, and Tom van Engers. Integrating CP-Nets in Reactive BDI Agents. In Matteo Baldoni, Mehdi Dastani, Beishui Liao, Yuko Sakurai, and Rym Zalila Wenkstern, editors, *PRIMA 2019: Principles and Practice of Multi-Agent Systems*, pages 305–320. Springer International Publishing, 2019.

[123] Mostafa Mohajeri Parizi, Giovanni Sileno, and Tom van Engers. Declarative Preferences in Reactive BDI Agents. In Takahiro Uchiya, Quan Bai, and Iván Marsá Maestre, editors, *PRIMA 2020: Principles and Practice of Multi-Agent Systems*, pages 215–230, Cham, 2021. Springer International Publishing.

[124] Mostafa Mohajeri Parizi, Giovanni Sileno, and Tom van Engers. Preference-based goal refinement in bdi agents. In *Proceedings of the 21st International Conference on Autonomous Agents and Multiagent Systems*, AAMAS '22, page 917–925, Richland, SC, 2022. International Foundation for Autonomous Agents and Multiagent Systems.

[125] Mostafa Mohajeri Parizi, Giovanni Sileno, and Tom van Engers. Seamless integration and testing for mas engineering. In *Engineering Multi-Agent Systems*, pages 254–272, Cham, 2022. Springer International Publishing.

[126] Mostafa Mohajeri Parizi, Giovanni Sileno, Tom van Engers, and Sander Klous. Run, agent, run! architecture and benchmarking of actor-based agents. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE 2020, page 11–20, New York, NY, USA, 2020. Association for Computing Machinery.

[127] Mailyn Moreno, Juan Pavón, and Alejandro Rosete. Testing in agent oriented methodologies. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5518 LNCS(PART 2):138–145, 2009.

[128] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, Ltd, 2012.

[129] Cu D. Nguyen, Anna Perini, Carole Bernon, Juan Pavón, and John Thangarajah. Testing in multi-agent systems. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6038 LNCS:180–190, 2011.

[130] Helen Nissenbaum. Privacy as contextual integrity. *Wash. L. Rev.*, 79:119, 2004.

[131] Lin Padgham and Dhirendra Singh. Situational preferences for BDI plans. *Proceedings of the 2013 international conference ...*, pages 1013–1020, 2013.

[132] Lin Padgham and Michael Winikoff. *Developing intelligent agent systems: A practical guide*, volume 13. John Wiley & Sons, 2004.

[133] Stipe Pandžić, Jan Broersen, and Henk Aarts. BOID*: Autonomous goal deliberation through abduction. In *Proceedings of the 21st International Conference on Autonomous Agents and Multiagent Systems*, AAMAS '22, page 1019–1027, Richland, SC, 2022. International Foundation for Autonomous Agents and Multiagent Systems.

[134] Mostafa Mohajeri Parizi, L. Thomas van Binsbergen, Giovanni Sileno, and Tom van Engers. A modular architecture for integrating normative advisors in mas. In *Multi-Agent Systems*, pages 312–329, Cham, 2022. Springer International Publishing.

[135] Gabriella Pigozzi, Alexis Tsoukiàs, and Paolo Viappiani. Preferences in artificial intelligence. *Annals of Mathematics and Artificial Intelligence*, 77(3-4):361–401, 2016.

[136] John Pourdehnad, Kambiz Maani, and Habib Sedehi. System dynamics and intelligent agent-based simulation: where is the synergy. In *Proceedings of the XX International Conference of the System Dynamics society*, 2002.

[137] David Poutakidis, Michael Winikoff, Lin Padgham, and Zhiyong Zhang. *Debugging and Testing of Multi-Agent Systems using Design Artefacts*, pages 215–258. Springer US, 2009.

[138] Felixie Rafalimanana, Jean Razafindramintsa, Josué Ratovondrahona, Mahatody Thomas, and Victor Manantsoa. Publish a jason agent bdi capacity as web service rest and soap. In *Proceedings of the 8th International Conference on Sciences of Electronics, Technologies of Information and Telecommunications (SETIT'18), Vol.1*, pages 163–171, 01 2020.

[139] Anand S Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In Walter de Velde and John W Perram, editors, *Agents Breaking Away*, pages 42–55, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[140] Anand S Rao, Michael P Georgeff, and others. BDI agents: From theory to practice. *Icmas*, 95:312–319, 1995.

[141] Sebastian Rodriguez, Nicolas Gaud, and Stéphane Galland. Sarl: A general-purpose agent-oriented programming language. In *2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT)*, volume 3, pages 103–110, 2014.

[142] Sebastian Sardina and Lin Padgham. A BDI agent programming language with failure handling, declarative goals, and planning. *Autonomous Agents and Multi-Agent Systems*, 23(1):18–70, 2011.

[143] H.J. Scholl. Agent-based and system dynamics modeling: a call for cross study and joint research. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*, pages 8 pp.–, 2001.

[144] John R. Searle. *Speech acts: An Essay in the Philosophy of Language.* Cambridge University Press, 1969.

[145] Sara Shakeri, Valentina Maccatrozzo, Lourens Veen, Rena Bakhshi, Leon Gommans, Cees De Laat, and Paola Grosso. Modeling and matching digital data marketplace policies. In *IEEE 15th International Conference on eScience, eScience 2019*, pages 570–577. Institute of Electrical and Electronics Engineers Inc., 9 2019.

[146] Smadar Shilo, Hagai Rossman, and Eran Segal. Axes of a revolution: challenges and promises of big data in healthcare. *Nature Medicine*, 26:29–38, 1 2020.

[147] Yoav Shoham. Agent-oriented programming. *Artificial intelligence*, 60(1):51–92, 1993.

[148] Giovanni Sileno. *Aligning Law and Action.* PhD thesis, University of Amsterdam, 2016.

[149] Giovanni Sileno, Alexander Boer, and Tom van Engers. Revisiting Constitutive Rules. In *Proceedings of the 6th Workshop on Artificial Intelligence and the Complexity of Legal Systems (AICOL 2015)*, 2015.

[150] Giovanni Sileno, L. Thomas van Binsbergen, Matteo Pascucci, and Tom van Engers. DPCL: a language template for normative specifications. *Workshop on Programming Languages and the Law (ProLaLa 2022), co-located with POPL 2022*, 2020.

[151] Lavindra De Silva. BDI Agent Reasoning with Guidance from HTN Recipes. *AAMAS '17 Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, pages 759–767, 2017.

[152] Dhirendra Singh, Sebastian Sardina, Lin Padgham, and Geoff James. Integrating learning into a bdi agent for environments with changing dynamics. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Three*, IJCAI'11, page 2525–2530. AAAI Press, 2011.

[153] Dhirendra Singh, Sebastian Sardina, Lin Padgham, and Geoff James. Integrating learning into a BDI agent for environments with changing dynamics. *IJCAI International Joint Conference on Artificial Intelligence*, pages 2525–2530, 2011.

[154] Agnieszka Szpak. Legality of Use and Challenges of New Technologies in Warfare – the Use of Autonomous Weapons in Contemporary or Future Wars. *European Review*, 28(1):118–131, feb 2020.

[155] Stanisław Szufa, Piotr Faliszewski, Piotr Skowron, Arkadii Slinko, and Nimrod Talmon. Drawing a map of elections in the space of statistical cultures. In *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '20, page 1341–1349, Richland, SC, 2020. International Foundation for Autonomous Agents and Multiagent Systems.

[156] Mihnea Tufis and Jean-Gabriel Ganascia. Grafting norms onto the BDI agent model. *A Construction Manual for Robots' Ethical Systems. Cognitive Technologies*, 2015.

[157] L. Thomas van Binsbergen, Milen G. Kebede, Joshua Baugh, Tom van Engers, and Dannis G. van Vuurden. Dynamic generation of access control policies from social policies. *Procedia Computer Science*, 198:140–147, 2022. 12th International Conference on Emerging Ubiquitous Systems and

Pervasive Networks / 11th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare.

[158] L. Thomas van Binsbergen, Lu-Chi Liu, Robert van Doesburg, and Tom van Engers. eFLINT: A domain-specific language for executable norm specifications. *GPCE 2020 - Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, Co-located with SPLASH 2020*, pages 124–136, 2020.

[159] Tom van Engers and Erwin Glassee. Facilitating the legislation process using a shared conceptual model. *IEEE Intelligent Systems*, 16(1):50–58, 2001.

[160] Lourens E Veen, Sara Shakeri, and Paola Grosso. Mahiru: a federated, policy-driven data processing and exchange system. *arXiv preprint arXiv:2210.17155*, 2022.

[161] Simeon Visser, John Thangarajah, and James Harland. Reasoning about preferences in intelligent agent systems. *IJCAI International Joint Conference on Artificial Intelligence*, pages 426–431, 2011.

[162] Simeon Visser, John Thangarajah, James Harland, and Frank Dignum. Preference-based reasoning in BDI agent systems. *Autonomous Agents and Multi-Agent Systems*, 30(2):291–330, 2016.

[163] Wietske Visser, Reyhan Aydoğan, Koen V. Hindriks, and Catholijn M. Jonker. A framework for qualitative multi-criteria preferences. *ICAART 2012 - Proceedings of the 4th International Conference on Agents and Artificial Intelligence*, 1:243–248, 2012.

[164] Wietske Visser, Koen V. Hindriks, and Catholijn M. Jonker. Goal-based qualitative preference systems. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7784 LNAI:153–169, 2013.

[165] Wolff Heintschel von Heinegg. Considerations of Necessity under Article 57(2)(a)(ii), (c), and (3) and Proportionality under Article 51(5)(b) and Article 57(2)(b) of Additional Protocol I. In Claus Kreß and Robert Lawless, editors, *Necessity and Proportionality in International Peace and Security Law*, pages 325–342. Oxford University Press, Oxford, nov 2020.

[166] Nic Wilson. Extending CP-Nets with Stronger Conditional Preference Statements. In *19th National Conference on Artificial Intelligence (AAAI'04)*, pages 735–741, 2004.

[167] Michael Winikoff. BDI agent testability revisited. *Autonomous Agents and Multi-Agent Systems*, 31(5):1094–1132, 2017.

[168] Michael Winikoff and Stephen Cranefield. On the testability of BDI agent systems. *IJCAI International Joint Conference on Artificial Intelligence*, 2015-Janua:4217–4221, 2015.

[169] Michael Winikoff, Louise Dennis, and Michael Fisher. *Slicing Agent Programs for More Efficient Verification*, volume 11375 LNAI. Springer International Publishing, 2019.

[170] Michael Winikoff, Lin Padgham, James Harland, and John Thangarajah. Declarative & procedural goals in intelligent agent systems. In *8th International Conference on Principles of Knowledge Representation and Reasoning*, KR'02, page 470–481, 2002.

[171] Lu Zhang, Reginald Cushing, Leon Gommans, Cees De Laat, and Paola Grosso. Modeling of Collaboration Archetypes in Digital Market Places. *IEEE Access*, 7:102689–102700, 2019.

[172] Zhiyong Zhang, John Thangarajah, and Lin Padgham. Automated unit testing for agent systems. In *ENASE 2007 - Proceedings of the 2nd International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 10–18, 2007.

[173] Xin Zhou, Adam Belloum, Michael H Lees, Tom van Engers, and Cees de Laat. Costly incentives design from an institutional perspective: cooperation, sustainability and affluence. *Proceedings of the Royal Society of London Series A*, 478(2265):20220393, 2022.

[174] Xin Zhou, Reginald Cushing, Ralph Koning, Adam Belloum, Paola Grosso, Sander Klous, Tom van Engers, and Cees de Laat. Policy enforcement for secure and trustworthy data sharing in multi-domain infrastructures. In *2020 IEEE 14th International Conference on Big Data Science and Engineering (BigDataSE)*, pages 104–113. IEEE, 2020.

[175] T Zurek and M Araszkiewicz. Modeling teleological interpretation. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Law*, pages 160–168. ACM, 2013.

[176] Tomasz Zurek. Modeling conflicts between legal rules. In M. Ganzha, L. Maciaszek, and M. Paprzycki, editors, *Proceedings of the 2016 Federated Conference on Computer Science and Information Systems*, volume 8 of *Annals of Computer Science and Information Systems*, pages 393–402. IEEE, 2016.

[177] Tomasz Zurek. Goals, values, and reasoning. *Expert Systems with Applications*, 71:442–456, 2017.

[178] Tomasz Zurek, Mostafa Mohajeri Parizi, Jonathan Kwik, and Tom van Engers. Can a military autonomous device follow international humanitarian law? In *Legal Knowledge and Information Systems*, pages 273–278. IOS Press, 2022.

Samenvatting

Met de toenemende digitalisering van de samenleving wordt een steeds groter deel van ons dagelijks leven beïnvloed door softwaresystemen die beslissingen ondersteunen of automatiseren (over visumaanvragen, sollicitaties, kredietplaatsingen, hypotheken, verzekeringen, . . .), door middel van op regels gebaseerde, datagestuurde of hybride methoden. Deze transformatie vormt de motivatie voor de in dit proefschrift beschreven technische benaderingen om het ontwerp, de ontwikkeling, de implementatie en het onderhoud van digitale systemen te regelen in overeenstemming met de regelgeving. In dit proefschrift concentreren we ons op computationele agenten en hun interactie met normen, om een reeks ontwerp- en modelleringstools, computationele methodologieën en inzichten met betrekking tot het beheer van sociaal-technische systemen te ontwerpen en te ontwikkelen. Agentmodellen kunnen worden gebruikt om de verschillende belanghebbenden vast te leggen die deelnemen aan het sociale systeem, elk met hun eigen soort overtuigingen, intenties en voorkeuren. Normmodellen stellen verwachtingen en dus criteria voor de evaluatie van gedrag; als zodanig vormen ze een fundamentele basis voor sociale coördinatiemechanismen en voor oordelen over naleving.

Dit proefschrift richt zich op agentmodellering en de bijbehorende modeluitvoering als een belangrijke stap in systeemontwerp en beleidsvormingsactiviteiten met betrekking tot digitale infrastructuren. Het introduceert een agentgebaseerd programmeerraamwerk (genaamd ASC2) op basis van de believe-desire-intention (BDI), naast een schaalbaar multi-agentsysteem als runtime-omgeving. Het uitvoeringsmodel van de agents is gebaseerd op het actormodel en het framework wordt geanalyseerd en vergeleken met andere state-of-the-art agent-georiënteerde programmeerframeworks. Het proefschrift gaat verder in op de waarde van het gebruik van reguliere softwareontwikkelingstools in agent-based programmeren en illustreert dit door ASC2 te combineren met andere tools zoals testing, debugging en zelfs DevOps-tools.

Op taalniveau worden, om de transparantie van de modellen en de besluitvorming van de agent te vergroten, contextafhankelijke voorkeuren, zowel procedureel

als declaratief, aan het raamwerk toegevoegd. Verder wordt een methode gepresenteerd om voorkeuren te gebruiken voor het verfijnen van abstracte doelen, waarbij de how-to-do kennis van de agenten wordt uitgebreid met what-to-do kennis. Ten slotte wordt een modulaire, op BDI gebaseerde architectuur geïntroduceerd voor het integreren van normen in MAS via normatieve adviseurs. Deze architectuur stelt de agenten in staat om normen over te nemen, te laten vallen en ermee te redeneren door ze te integreren in hun redeneringscyclus zonder de autonomie van de agent te beperken. De normatieve adviseursarchitectuur houdt rekening met regulerende en constitutieve normencategorieën, persoonlijke en groepsnormen, verschillende operationaliseringen van normen op basis van gecentraliseerde, gelokaliseerde en ad-hocarchitecturen, en ook met de aanwezigheid van meerdere bronnen van normen, waaronder complementaire en/of tegenstrijdige bronnen.

Om de mogelijkheden van de voorgestelde benaderingen te demonstreren, worden twee verschillende voorbeeldcasus gepresenteerd. De eerste casestudy richt zich op het gebruik van ad-hoc-normen als coördinatiemiddel in een normatief multi-agentsysteem, en de tweede casestudy richt zich op het omschrijven van acties van autonome agenten met gecodeerde wetten.

Abstract

With the increasing digitization of our society, more of our daily life is being affected by software systems supporting us taking (partially) automating decisions (about visa applications, job applications, credit placements, mortgages, insurances, ...), by means of rule-based, data-driven, or hybrid methods. This transformation motivates the introduction of engineering approaches to govern the design, development, deployment, and maintenance of digital systems in alignment with regulations. This dissertation tackles this challenge by focusing on computational agents, and their interaction with norms, to conceive and develop a set of design and modelling tools, computational methodologies, and insights concerning the governance of socio-technical systems. Agent models can be utilized to capture the different stakeholders participating in the social system, each with their own type of beliefs, intentions, preferences. Norm models sets expectations and thus criteria for the evaluation of behaviors; as such, they form a fundamental basis for social coordination mechanisms and for judgments about compliance.

This dissertation focuses on agent modelling and the associated model-execution as a principal step in system design and policy-making activities on digital infrastructures. It introduces an agent-based programming framework (named ASC2) based on the belief-desire-intention (BDI), alongside a scalable multi-agent system as runtime environment. The execution model of the agents is based on the actor model and the framework is analysed and benchmarked against other state-of-the-art agent-oriented programming frameworks. The dissertation further elaborates on the value of utilizing mainstream software development tools in agent-based programming and illustrates this by bridging ASC2 with multiple tools such as testing, debugging and even DevOps tools.

At the level of language, to increase the transparency of the models and agent's decision-making, context-dependent preferences, both procedural and declarative are added to the framework. Furthermore, a method is presented to utilize preferences for refinement of abstract goals, extending the how-to-do knowledge of the agents with what-to-do knowledge. Finally, a modular BDI-based architecture is

introduced for integrating norms into MAS via normative advisors. This architecture allows for the agents to adopt, drop and reason with norms by integrating them in their reasoning cycle whilst not constraining the agent's autonomy. The normative advisor architecture considers regulative and constitutive categories of norms, personal and group norms, different operationalizations of norms based on centralized, localized, and ad-hoc architectures, and also presence of multiple sources of norms, including complementing and/or contradicting ones.

To fully demonstrate the capabilities of the proposed approaches, two different example cases are presented. The first case study focuses on the utilization of ad-hoc norms as means of coordination in a normative multi-agent system, and the second case study focuses on circumscribing actions of autonomous agents with encoded laws.

ISBN: 978-94-6473-443-0