Knowledge Engineering and Machine Learning Group
UNIVERSITAT POLITÈCNICA DE CATALUNYA

Introducing Norms into Practical Reasoning Agents

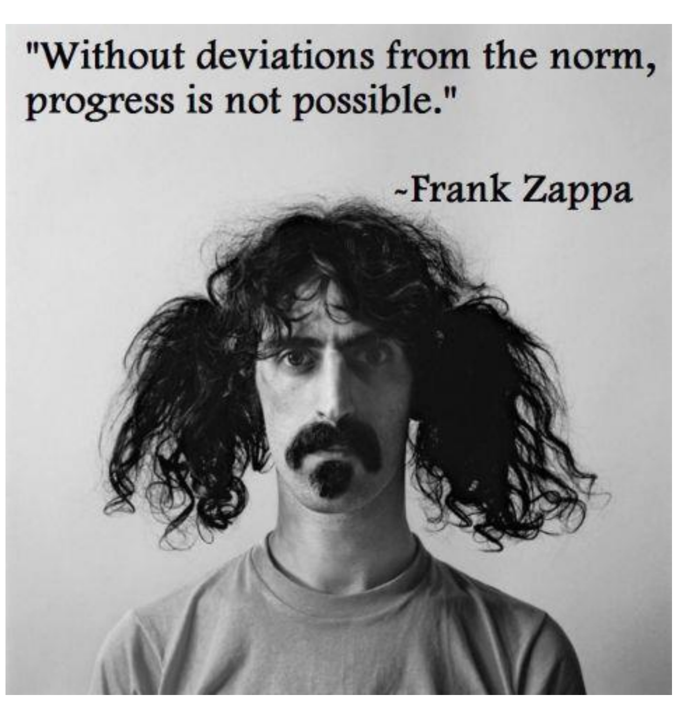
by

Sofia Panagiotidi

Advisor: Javier Vázquez-Salceda

Departament de Llenguatges i Sistemes Informàtics
Artificial Intelligence PhD Program

July 2014


"Without deviations from the norm,
progress is not possible."
-Frank Zappa

Abstract

As distributed electronic systems grow to include thousands of components, from grid to peer-to-peer nodes, from (Semantic) Web services to web-apps to computation in the cloud, governance of such systems is becoming a real challenge. Modern approaches ensuring appropriate individual entities' behaviour in distributed systems, which comes from multi-agent systems (MAS) research, use *norms* (or regulations or policies) and/or communication protocols to express a different layer of desired or undesired states. From the individual's perspective, an agent needs to be able to function in an environment where norms act as behavioural restrictions or guidelines as to what is appropriate, not only for the individual but also for the community.

In the literature the concept of norms has been defined from several perspectives: as a rule or standard of behaviour shared by members of a social group, as an authoritative rule or standard by which something is judged, approved or disapproved, as standards of right and wrong, beauty and ugliness, and truth and falsehood, or even as a model of what should exist or be followed, or an average of what currently does exist in some context. Currently there exist in the literature: 1) some treatments that formally connect the deontic aspects of norms with their operationalisation; 2) some treatments that properly distinguish between abstract norms and their (multiple) instantiations at runtime; 3) little work that formalises the operational semantics in a way that ensures flexibility in their translation to actual implementations while ensuring unambiguous interpretations of the norms; 4) little work that is suitable for both institutional-level norm monitoring and individual agent norm-aware reasoning to ensure that both are aligned; 5) few works that explore how the norms may affect the decision making process of an agent when the process includes planning mechanisms at runtime for means-ends reasoning. However, currently there is no work that includes both a formalism and an implementation covering 1-5 altogether.

This thesis presents work towards the above five areas. We give a proposal to bridge the gap between a single norm formalisation and the actual mechanisms used for norm-aware planning, in order to create a normative practical reasoning mechanism. One way to do this is by reducing deontic-based norm definitions to temporal logic formulas which, in turn, can be translated into planning operational semantics. Based

on these semantics, we create a mechanism to support practical normative reasoning that can be used by agents to produce and evaluate their plans. We construct a norm-oriented agent that takes into consideration operationalised norms during the plan generation phase, using them as guidelines to decide the agent's future action path. To make norms influence plan generation, our norm operational semantics is expressed as an extension of the planning domain, acting as a form of temporal restrictions over the trajectories (plans) computed by the planner. We consider two approaches to do so. One implementing the semantics by using planning with constraints through paths and the other by directly translating the norms into domain knowledge to be included into the planning domain. We explore a scenario based on traffic laws in order to demonstrate the usability of our proposal. We also show how our normative frameworks are successfully integrated into an existing BDI agent implementation, 2APL. For each approach taken, we present quantitative experimental results and illustrate the opportunities for further research.

Resumen

La gestión de sistemas electrónicos distribuidos se está convirtiendo en un auténtico reto a medida que dichos sistemas crecen incluyendo múltiples componentes, desde nodos grid a peer-to-peer, servicios de la Web semántica, aplicaciones web o computación en la nube. Los enfoques modernos que aseguran un comportamiento adecuado de las entidades individuales en sistemas distribuidos, y que provienen de la investigación en sistemas multi-agentes (MAS), utilizan *normas* (o regulaciones o políticas) y/o protocolos de comunicación para expresar un nivel diferente de estados deseados o no deseados. Desde la perspectiva del individuo, un agente necesita poder funcionar en un entorno donde las normas actúen como restricciones o directrices de comportamiento respecto a lo que es apropiado, no únicamente para el individuo sino para la comunidad en su conjunto.

En la literatura el concepto de norma se ha definido desde diferentes perspectivas: como una regla o estándar de comportamiento compartida por los miembros de un grupo social, como una regla autoritaria o estándar por el cual se juzga, se aprueba o desaprueba, como estándar de lo correcto o incorrecto, belleza o fealdad, verdad o falsedad, o incluso, como un modelo que debería existir o ser seguido, o como un promedio de lo que actualmente existe en un contexto determinado. En la actualidad se pueden encontrar en la literatura: 1) trabajos que conectan formalmente los aspectos deónticos de las normas con su operacionalización; 2) trabajos que distinguen adecuadamente entre normas abstractas y sus (múltiples) instanciaciones en tiempo de ejecución; 3) algún ejemplo que formaliza las semánticas operacionales de manera que se asegura la flexibilidad en su traducción a implementaciones garantizando a su vez interpretaciones no ambiguas de las normas; 4) algún trabajo que se adecúa tanto a la monitorización de normas a nivel institucional como al razonamiento basado en normas a nivel de los agentes individuales y que asegura que ambos están alineados; 5) algún trabajo que explora como las normas pueden afectar al proceso de toma de decisiones de un agente cuando el proceso incluye mecanismos de planificación en tiempo real para un razonamiento medios-fines. Sin embargo, actualmente no existe ningún enfoque que incluya formalismos e implementaciones abordando los 5 puntos al mismo tiempo.

La presente tesis propone contribuciones en las cinco áreas mencionadas. Se presenta una propuesta para establecer un enlace entre la formalización de una norma y los mecanismos utilizados en la planificación basada en normas con el objetivo de crear un mecanismo de razonamiento práctico normativo. Una forma de conseguirlo es mediante la reducción de las definiciones de normas basadas en deóntica a fórmulas de lógica temporal que, a su vez, pueden ser traducidas a semánticas operacionales de planificación. Basándose en estas semánticas, se ha creado un mecanismo para dar soporte al razonamiento normativo práctico que puede ser utilizado por los agentes para producir y evaluar sus planes. Se ha construido un agente orientado a normas que tiene en consideración las normas operacionalizadas durante la fase de generación de planes, utilizándolas como directrices para decidir el futuro curso de acción del agente. Para conseguir que las normas influyan en la generación de planes, nuestras semánticas operacionales de normas se expresan como una extensión del dominio de la planificación, actuando como una forma de restricciones temporales sobre las trayectorias (planes) computadas por el planificador. Se han considerado dos enfoques para realizarlo. Uno, implementando las semánticas utilizando planificación con restricciones a través de caminos y otro, traduciendo directamente las normas en conocimiento del dominio que se incluirá en el dominio de planificación. Se explora un escenario basado en normas de circulación de tráfico para demostrar la usabilidad de nuestra propuesta. Se mostrará también como nuestro marco normativo se integra satisfactoriamente en una implementación existente de agentes BDI, 2APL. Para cada enfoque considerado, se presentan resultados experimentales cuantitativos y se ilustran las oportunidades para futuros trabajos de investigación.

Resum

A mesura que els sistemes electrònics distribuïts creixen per incloure milers de components, des de nodes grid a peer-to-peer fins a serveis de la Web semàntica, aplicacions web o computació al núvol, la gestió d'aquests sistemes s'està convertint en un autèntic repte. Els enfocs moderns que asseguren el comportament apropiat de les entitats individuals en sistemes distribuïts, que prové de la recerca en sistemes multi-agents, utilitzen *normes* (o regulacions o polítiques) i/o protocols de comunicació per expressar una capa diferent d'estats desitjats o no desitjats. Des de la perspectiva de l'individu, un agent necessita poder funcionar en un entorn on les normes actuïn com a restriccions de comportament o guies respecte al que és apropiat, no només per al individu sinó per a la comunitat.

En la literatura el concepte de normes s'ha tractat des de diferents perspectives: com una regla o estàndard de comportament compartida pels membres d'un grup social, com una regla o estàndard autoritari pel qual alguna cosa és jutjada, aprovada o desaprovada, com estàndard del correcte i del incorrecte, bellesa i lletjor, veritat i falsedat, o inclús com un model del que hauria d'existir o ser seguit, o com una mitjana del que actualment existeix en un context donat. Actualment trobem en la literatura: 1) alguns tractaments que connecten formalment els aspectes deòntics de les normes amb la seva operacionalització; 2) alguns tractaments que distingeixen adequadament entre normes abstractes i les seves (múltiples) instanciacions en temps real; 3) alguns exemples que formalitzen les semàntiques operacionals de manera que asseguren flexibilitat en la seva traducció a implementacions garantint interpretacions no ambigües de les normes; 4) alguns treballs adequats per a la monitorització de normes a nivell institucional i per al raonament basat en normes en agents individuals assegurant que ambdós estan alineats; 5) alguns treballs que exploren com les normes poden afectar el procés de presa de decisions d'un agent quan el procés inclou mecanismes de planificació en temps real per a raonament mitjans-finalitats. D'altra banda, actualment no existeix cap enfoc que inclogui formalismes i implementacions cobrint els punts 1-5 a la vegada.

Aquesta tesi presenta contribucions en les cinc àrees esmentades. Presentem una proposta per establir un enllaç entre la formalització d'una norma i els mecanismes

emprats en la planificació basada en normes per tal de crear un mecanisme de raonament pràctic normatiu. Una manera d'aconseguir-ho és reduint les definicions de normes deòntiques a fórmules de lògica temporal les quals poden ser traduïdes a semàntiques de planificació operacional. Basant-nos en aquestes semàntiques, hem creat un mecanisme per donar suport al raonament normatiu pràctic que pot ser emprat per agents per produir i avaluar els seus plans. Hem construït un agent orientat a normes que pren en consideració durant la fase de generació de plans les normes operacionalitzades, utilitzant-les com a guia per decidir el futur curs d'acció de l'agent. Per tal de fer que les normes influenciïn la generació de plans, les nostres semàntiques operacionals de normes s'expressen com una extensió del domini de la planificació, actuant com una mena de restriccions temporals sobre les trajectòries (plans) computades pel planificador. Considerem dos enfocs per dur-ho a terme. Un implementant les semàntiques emprant planificació amb restriccions per mitjà de camins i l'altre traduint directament les normes en coneixement del domini a ser inclòs en el domini de planificació. Explorem un escenari basat en les normes de circulació de tràfic per demostrar la usabilitat de la nostra proposta. Mostrarem també com el nostre marc normatiu s'integra satisfactòriament en una implementació existent d' agent BDI, 2APL. Per cada enfoc considerat, presentem resultats experimentals quantitatius i il·lustrem les oportunitats per treballs de recerca futurs.

Περίληψη

Καθώς τα ηλεκτρονικά συστήματα στις μέρες μας φτάνουν να περιλαμβάνουν χιλιάδες στοιχεία, από grid μέχρι peer-to-peer δίκτυα, από (σημασιολογικές) υπηρεσίες έως web-apps στο Cloud, η διακυβέρνηση τέτοιων συστημάτων γίνεται ολοένα και πιο δύσκολη. Σύγχρονες μέθοδοι που προέρχονται από έρευνα πάνω σε πολυπρακτορικά συστήματα χρησιμοποιούν κανονισμούς (νόρμες) και/ή πρωτόκολλα επικοινωνίας για να εκφράσουν ένα επίπεδο επιθυμητών ή ανεπιθύμητων καταστάσεων. Από την οπτική του ατόμου, ένας πράκτορας είναι απαραίτητο να μπορεί να λειτουργεί σε ένα περιβάλλον όπου οι κανονισμοί δρουν ως συμπεριφορικοί περιορισμοί ή υποδείξεις ως προς το τι είναι θεμιτό, όχι μόνο για το άτομο, αλλά επιπλέον και για την κοινότητα.

Στη βιβλιογραφία η έννοια του κανονισμού έχει οριστεί από διαφορετικές οπτικές: ως κοινό πρότυπο συμπεριφοράς για τα μέλη μίας κοινωνικής ομάδας, ως εξουσιαστικός κανόνας ή ως πρότυπο μέσω του οποίου κάτι κρίνεται, εγκρίνεται ή αποδοκιμάζεται, ως πρότυπο σωστού και λάθους, ομορφιάς και ασχήμιας, αλήθειας ή ψεύδους, ή ακόμη και ως πρότυπο του τι θα μπορούσε να υπάρχει ή να ακολουθείται, ή ως μέσος όρος του τι υπάρχει επί του παρόντος μέσα σε ένα πλαίσιο. Έως σήμερα στη βιβλιογραφία υπάρχουν: 1) κάποιες εργασίες οι οποίες επισήμως συνδέουν τη δεοντολογική διάσταση των κανονισμών με την λειτουργία τους στην πράξη· 2) κάποιες έρευνες οι οποίες επισήμως διαφοροποιούν τους αφηρημένους κανονισμούς από τις (πολλαπλές) ενσαρκώσεις τους σε πραγματικό χρόνο· 3) λίγη έρευνα η οποία επισημοποιεί τη λειτουργική τους σημασιολογία με τέτοιο τρόπο ώστε να εξασφαλίζεται η προσαρμοστικότητα κατά τη μετάφρασή τους σε πραγματικές υλοποιήσεις, ενώ ταυτόχρονα να εξασφαλίζεται και η μη διφορούμενη ερμηνεία των κανονισμών· 4) λίγες εργασίες οι οποίες είναι κατάλληλες ταυτόχρονα για επιτήρηση και εφαρμογή των κανονισμών σε θεσμικό επίπεδο καθώς και για λήψη αποφάσεων σε ατομικό επίπεδο κάθε πράκτορα, εξασφαλίζοντας οτι τα δύο είναι εναρμονισμένα· 5) λίγες εργασίες οι οποίες ερευνούν πώς οι κανονισμοί μπορούν να επηρεάσουν τη διαδικασία λήψης αποφάσεων ενός πράκτορα όταν η διαδικασία περιλαμβάνει μηχανισμούς παραγωγής πλάνων (planning) σε πραγματικό χρόνο προς επίτευξη ορισμένων στόχων. Παρ΄ όλα αυτά, δεν υπάρχει καμία έρευνα που να περιλαμβάνει ένα φορμαλισμό και μία υλοποίηση που να καλύπτει και τα 5 μαζί.

Αυτή η εργασία παρουσιάζει έργο προς τις πέντε παραπάνω κατευθύνσεις. Παραθέτουμε μια πρόταση για να γεφυρώσουμε το χάσμα μεταξύ ενός συγκεκριμένου φορμαλισμού των

κανονισμών και των μηχανισμών που χρησιμοποιούνται στο planning βάσει κανονισμών, ώστε να κατασκευάσουμε έναν πρακτικό μηχανισμό αποφάσεων με κανονισμούς. Ένας τρόπος να γίνει αυτό είναι μετατρέποντας τους δεοντολογικούς ορισμούς κανονισμών σε λογικές χρονικές φόρμουλες οι οποίες, με τη σειρά τους, μπορούν να μεταφραστούν σε πρακτικό φορμαλισμό planning. Βασιζόμενοι σε αυτό το φορμαλισμό, χτίζουμε ένα εργαλείο που υποστηρίζει έναν πρακτικό μηχανισμό αποφάσεων με κανονισμούς το οποίο μπορεί να χρησιμοποιηθεί από πράκτορες για να κατασκευάσουν και αξιολογήσουν τα πλάνα τους. Κατασκευάζουμε έναν πράκτορα κατευθυνόμενο από κανονισμούς, ο οποίος λαμβάνει υπόψην του πρακτικούς κανονισμούς κατά τη διάρκεια του υπολογισμού των πλάνων του, χρησιμοποιώντας τους ταυτόχρονα ως κατευθύνσεις για να αποφασίσει τις επόμενες κινήσεις του. Για να επηρεάσουν οι κανονισμοί τα πλάνα που παράγονται, ο φορμαλισμός μας εκφράζεται ως επέκταση του πεδίου planning, δρώντας ως κάποια μορφή λογικών περιορισμών πάνω στα πλάνα (μονοπάτια) που υπολογίζονται από τον planner. Εξετάζουμε δύο προσεγγίσεις γι' αυτό. Μία, υλοποιώντας το φορμαλισμό χρησιμοποιώντας planning με περιορισμούς πάνω στα μονοπάτια και άλλη μία, απευθείας μεταφράζοντας τους κανονισμούς σε όρους του πεδίου γνώσης (planning domain) ώστε να ενσωματωθούν στο πεδίο planning. Κατασκευάζουμε και μελετούμε ένα σενάριο βασισμένο στον κώδικα οδικής κυκλοφορίας για να επιδείξουμε τη χρησιμότητα της πρότασής μας. Επιπλέον, επιδεικνύουμε πως η κατασκευή μας μπορεί να ενσωματωθεί σε μια ήδη υπάρχουσα υλοποίηση BDI πρακτόρων, ονομαζόμενη 2APL. Για κάθε προσέγγιση, παρουσιάζουμε ποσοτικά πειράματικά αποτελέσματα και αναλύουμε τις ευκαιρίες για περαιτέρω έρευνα.

Acknowledgements

While I'm proud to call this PhD thesis my own, I have the greatest pleasure in acknowledging the intellectual and emotional support of friends, colleagues and family.

The process has sometimes seemed never-ending, and my first and greatest debt of gratitude must go to my advisor, Javier Vázquez, who patiently guided me through the challenges and difficulties that I faced. With his focused way of thinking he kept my ideas on track, and his enthusiasm and detailed revisions were a big part of making it through.

Sergio Álvarez also deserves my warmest appreciation. From the beginning till the very end, with his brilliance and generosity, he managed to solve countless seemingly unsolvable technical issues within astonishingly short time-periods. I counted on you Sergio, and you never stopped impressing me.

Ulises Cortés acted as a driving force throughout the thesis. He pushed me at times when I really needed to advance, and with his decisiveness he gave me strength to move forward, and I thank him for this.

I want to give special credit to Frank Dignum who kindly hosted me as a visiting student for 3 months at the University of Utrecht. His guidance was very helpful making my stay especially fruitful.

My colleagues in Barcelona (Bea, Roberto, Guiem, Sonia, Cristian, Joao, Arturo, Anna, Jonathan, Luis, Darío, Ignasi and so many more) were sources of laughter and support and made our office the most joyful place to work. I will always cherish the ambience and great memories from our fiestas inside and outside our workplace. I especially thank Roberto for his hospitality and the occasional meals together.

I greatly owe my family; my mother, Μάτα, my brother, Περικλή and γιαγιά Σοφία, for the love and support they embraced me with, even during periods of struggle. What I am, I owe to you.

Finally, I'd like to thank Álvaro. He can't possibly understand how much of a help he has been. Over recent months, often from a great distance, he has cheered me up and allowed me to build dreams through this very lonely path.

"I will obey every law, or submit to the penalty."

Chief Joseph

Contents

Abstract . . . v

Resumen . . . vii

Resum . . . ix

Acknowledgements . . . xiii

List of Figures . . . xxi
List of Tables . . . xxiii
1 Introduction . . . 1
1.1 Problem Statement . . . 2
1.2 Aim of the Thesis . . . 3
1.3 Thesis Objectives . . . 4
1.4 Out of Scope Issues . . . 5
1.5 Claims . . . 6
1.6 Structure of the Thesis . . . 7
2 Related Work . . . 11
2.1 Agent Orientation . . . 11
2.1.1 Practical Reasoning in Agents . . . 12
2.1.1.1 The BDI model . . . 13
2.1.1.2 Practical Frameworks Implementing BDI agents . . . 15
2.1.2 Agent Planning . . . 18
2.1.2.1 Action Language Formalisms . . . 19
2.1.2.2 Practical Frameworks Implementing Action Languages . . . 21
2.1.2.3 Planning in BDI Agents . . . 23
2.2 Social Structures and Agent Societies . . . 24
2.2.1 Organisational Models . . . 26
2.2.1.1 The Concept of Role . . . 27
2.2.1.2 Organisational Frameworks . . . 28
2.2.2 Institutions . . . 30
2.2.2.1 Human vs. Electronic Institutions . . . 31
2.2.2.2 Norms . . . 33
2.2.2.2.1 Regulative vs. Constitutive Norms . . . 34
2.2.2.2.2 Institutional and Normative Power . . . 35
2.2.2.2.3 Languages to Express Regulative Norms . . . 36
2.2.2.2.4 Operational Semantics for Regulated Systems . . . 38

2.2.2.3 Institutional View: Normative Multi-Agent Systems 39
2.2.2.3.1 Basic Concepts in the Modelling of Normative Multi-Agent Systems 40
2.2.2.3.2 Institutional Models for Normative Multi-Agent Systems 41
2.2.2.3.3 Hybrid Organisational-Institutional Models for Normative Multi-Agent Systems 43
2.2.2.3.4 Verification in Normative Multi-Agent Systems . . 46
2.2.2.3.5 Monitoring Normative Status 47
2.2.2.3.6 Relevant Approaches Outside the Agent Community 49
2.2.2.4 Agent-view: Norm-based Agents 51
2.2.2.4.1 Agent Frameworks Focusing on Constitutive (Counts-as) Norms 53
2.2.2.4.2 BDI-based Normative Autonomous Agents 54
2.2.2.4.3 Rule-Based Normative Agents 57
2.2.2.4.4 Rational Agents Normative Reasoning with Uncertainty 58
2.2.2.4.5 Normative Autonomous Agents Using Planning . 59
2.2.2.4.6 Plan Labelling Frameworks 60
2.2.2.4.7 Action Language and Abductive-Based Approaches 61
2.3 Summary 62
3 Conceptual Framework and Architecture 63
3.1 Requirements Analysis 64
3.1.1 Functional Requirements 64
3.1.1.1 Agent Model 64
3.1.1.2 Domain Model 66
3.1.1.3 Norm Model 66
3.1.2 Non-Functional Requirements 68
3.1.3 Technical Decisions 70
3.2 Conceptual Framework 72
3.2.1 Context 75
3.2.1.1 Ontology and Concept 76
3.2.1.2 Terms 77
3.2.1.3 Formulas, Relation Atoms and State of Affairs (StateFormula) 78
3.2.2 Roles 84
3.2.3 Agents 84
3.2.4 Initial State 85
3.2.5 Actions 85
3.2.6 Norms 86
3.2.7 Plans 90
3.3 Architecture 92
3.3.1 Components 92
3.3.2 BDI Agent Structure 94
3.4 Norm Design 98
3.4.1 Norm Lifecycle and Norm Instances 98
3.4.2 Dynamics of Norms 102
3.4.3 Primary, Secondary Norms and Interaction Between Them 102

3.5 Discussion 103
4 Normative Practical Reasoning Using Temporal Control Rules 105
4.1 First-Order Linear Temporal Logic 107
4.2 Extensions of fo-LTL for norms 108
4.3 Formalisation 110
4.3.1 Norms 110
4.3.2 Norm Instances 112
4.3.3 Norm Lifecycle 113
4.3.4 From Norm to Norm Instances 117
4.4 Reduction to Deontic Logics 118
4.4.1 Reduction to Achievement Obligations 119
4.4.2 Reduction to Maintenance Obligations 120
4.4.3 Reduction to Dyadic Maintenance Obligations 121
4.5 Example 121
4.5.1 Pizza Delivery Domain 121
4.5.2 Norms 125
4.6 Planning with Norms 127
4.6.1 Plans and Actions 128
4.6.2 Types, Numeric Expressions, Conditions and Effects 128
4.6.3 Calculation of Plan Cost 129
4.6.4 The Normative Planning Problem 129
4.6.5 Implementation via a Modified TLPLAN Algorithm 131
4.6.6 Experimental Results 133
4.7 Discussion 137
4.7.1 Contributions and Extensions 137
4.7.2 Revisiting Requirements 139
5 Practical Normative Reasoning with Repair Norms and Integration into a BDI Agent 141
5.1 Formalisation 142
5.1.1 Norms 142
5.1.2 Norm Instances 147
5.1.3 Norm Lifecycle 147
5.1.4 From Norm to Norm Instances 148
5.2 Planning with Norms 149
5.2.1 Plans, Actions, and Plan Cost 150
5.2.2 The Normative Planning Problem 151
5.2.3 Implementation Rules for Norm Lifecycle 153
5.2.4 Implementation Rules for Normative Planning Problem 157
5.2.5 Computational Overhead 158
5.2.6 Results 159
5.2.6.1 Tools 159
5.2.6.2 Execution Results 160
5.3 Connecting the Normative Reasoner with the 2APL BDI Core 161
5.3.1 2APL 162
5.3.1.1 2APL Elements 162
5.3.1.2 2APL Deliberation Cycle 164
5.3.2 2APL with Embedded Norm-Aware Planner 165
5.3.2.1 Modified 2APL Lifecycle 165

5.3.2.2 General Architecture 167
5.3.3 Adapting Inputs Between 2APL and the Normative Planner 167
5.3.3.1 Adapting Inputs from 2APL into Normative Planner 168
5.3.3.2 Adapting the Normative Planner Output into 2APL 170
5.3.4 Running the Normative Agent in 2APL 171
5.4 Discussion 174
5.4.1 Contributions and Extensions 174
5.4.2 Revisiting Requirements 176
6 Conclusions 179
6.1 Contributions 180
6.2 Revisiting Claims 182
6.3 Extensions 183
6.3.1 General Extensions 183
6.3.2 Probabilistic Practical Normative Reasoning 184
Publications of the Author 187
Bibliography 189
A Basis Framework Semantics 209
A.1 Norms for Modelling Regulative Clauses 209
A.2 Formal Preliminaries 211
A.3 Structural Definitions 212
A.3.1 Agent Names and Roles 212
A.3.2 Norms 212
A.3.3 Instantiating Abstract Norms 214
A.4 Dynamic Semantics 215
A.5 Issues and Related Work 219
B Example Implementation Details 221
B.1 Pizza Delivery Example Models 221
B.2 Pizza Delivery Example TLPLAN Code 223
B.3 Pizza Delivery Example PDDL Code 230
B.4 Pizza Delivery Example 2APL Code 240
C Proofs 247
C.1 Achievement Obligations 247
C.1.1 Substitution for Achievement Obligations 248
C.1.2 Proof of K 248
C.1.3 Proof of Necessitation 248
C.2 Maintenance Obligations 249
C.2.1 Substitution for Maintenance Obligations 249
C.2.2 Proof of K 250
C.2.3 Proof of D 250
C.2.4 Proof of Necessitation 250
C.3 Dyadic Deontic Logic 251
C.3.1 Substitution for Dyadic Deontic Logic 252
C.3.2 Proof of K1 252
C.3.3 Proof of K2 253
C.3.4 Proof of K3 254
C.3.5 Proof of K4 255
Index of Terms 257

List of Figures

2.1 The PRS agent model (source: [Georgeff and Lansky, 1987]) 14
2.2 From human laws to their electronic representation 30

3.1 Overall Metamodel 74
3.2 Normative Metamodel 76
3.3 Context 76
3.4 Ontology 77
3.5 Concept 77
3.6 Term 77
3.7 Variable 78
3.8 Constant 78
3.9 Function 78
3.10 StateFormula 79
3.11 Atom 80
3.12 Atom 81
3.13 Negation 81
3.14 Conjunction 82
3.15 Disjunction 82
3.16 Implication 82
3.17 Universal Quantification 83
3.18 Existential Quantification 83
3.19 Roles 84
3.20 Role Set 84
3.21 Agents 85
3.22 Agent Set 85
3.23 Actions 86
3.24 Action Set 86
3.25 Norms 87
3.26 Norms Set 87
3.27 Plan 90
3.28 Action Grounding List 91
3.29 Action Grounding 91
3.30 Input Map 91
3.31 Framework architecture 93
3.32 Caption for LOF 99
3.33 Norm instance lifecycle with reparation and timeout handling 100

4.1 Self-loop alternating automaton-based norm instance lifecycle. 114

4.2 Norm Instance Timeline 118
4.3 Example Domain 122
4.4 Normative Planner using TLPLAN 132
4.5 Metric formula in TLPLAN 134
4.6 Route Solution 1 135
4.7 Route Solution 2 135
4.8 Route Solution 3 135
4.9 Route Solution 4 135

5.1 Layered norms (repairing each other) 144
5.2 PDDL pizza delivery domain example 151
5.3 Screenshot of the 2APL environment 163
5.4 The 2APL deliberation cycle 165
5.5 The modified 2APL deliberation cycle 166
5.6 2APL Planner Architecture 168
5.7 Abstract action represented by a PC-rule 171
5.8 2APL environment GUI. Delivery Map 172
5.9 2APL initial configuration for the pizza delivery example 173

A.1 Domain Environment and Normative Environment lifecycle 219

B.1 Normative model representation 221
B.2 Variables, constants, functions representation 222
B.3 Ontology representation 222
B.4 Some of the state formulas used in the preconditions and effects of the actions and in the conditions of the norms 223
B.5 The action DeliverPizza representation 223
B.6 norm3 representation 223
B.7 Pizza delivery example domain TLPLAN 228
B.8 Pizza delivery example problem TLPLAN 229
B.9 Pizza delivery example domain in PDDL 238
B.10 Pizza delivery example problem in PDDL 240
B.11 Pizza delivery example topology in separate 2APL file, topology.2apl . . 241
B.12 Pizza delivery agent in 2APL, main_agent.2apl 245

List of Tables

4.1 Actions description, expressed in the textual form of the metamodel in Section 3.2 . . . 124
4.2 Repair actions description, expressed in the textual form of the metamodel of Section 3.2 . . . 125
4.3 Example norms, expressed in the textual form of the metamodel of Section 3.2 . . . 126
4.4 Execution Results . . . 136
4.5 Requirements Analysis . . . 140

5.1 Example norms . . . 146
5.2 PDDL domain implementation rules . . . 156
5.3 Execution Results . . . 160
5.4 Belief Updates Transformation . . . 169
5.5 Beliefs Transformation . . . 170
5.6 Goals Transformation . . . 171
5.7 Execution Results in the 2APL agent environment . . . 174
5.8 Requirements Analysis . . . 177

In memory of my father.

Chapter 1

Introduction

With the growth of the Internet and the World Wide Web over the last twenty years, computational systems have become more and more complex resulting in highly complicated interconnected networks (e.g. grid nodes, peer-to-peer nodes, Web services, web-apps, mobile apps, sensor networks). One problem that system designers and managers are facing is how to design and implement such complex systems and tackle their increased complexity, while ensuring that the system behaves as expected.

One approach for governance of open distributed systems, which comes from multi-agent systems research, is to add social order mechanisms to the system, where the individual computational entities' behaviour is guided or even restricted in order to ensure certain behaviour. According to Paes et al. [de Barros Paes et al., 2006]:

"*Governance for open systems can be defined as the set of approaches that aim to establish and enforce some structure, set of norms or conventions that articulate or restrain interactions in order to make agents more effective in attaining their goals or more predictable.*"

Inspired by social theory and cognitive science, a fair amount of research in the area of Artificial Intelligence, and specifically in the domain of Intelligent Agents, designs computational models to represent social structures. Further to this, a substantial amount of work has been done on the development of sets of theories to model reasoning, plans and even emotions to construct more adaptable and evolvable software systems.

While this thesis concentrates on the practical aspects of building regulated systems, it uses elements from existing social models and theories. We introduce some of them in the following sections[1].

[1] A thorough analysis of existing models, theories and frameworks can be found in Chapter 2

1.1 Problem Statement

Organisational models increasingly play an important role in the development of larger and more complex distributed systems and service-oriented systems. As systems grow to include hundreds or thousands of components, it is essential to move from an agent-focused view of coordination and management to an organisation-focused one. The overall problem of analysing the different dimensions of agent organisations, and the co-evolution of agents and social structures in the organisation, requires a flexible approach with a solid theoretical foundation [Dignum, 2009].

In agent organisational models, goals and interactions are not specified in terms of the mental states of individual agents, but in terms of organisational concepts such as roles (or functions, or positions), groups (or communities), norms (or regulations or policies) and communication protocols (including ontologies). Often organisational approaches take a top-down perspective, i.e. organisational characteristics (goals, norms, etc.) fully determine the agent's plans, goals and rules [Padgham and Winikoff, 2004]. In these cases, agents are seen as actors that enact the role(s) described by the organisation design.

A common problem that occurs in the design and implementation of complex systems (both MAS or service-based systems) is the fact that specifications of the organisation of the system generally abstract from actual practice. That is, while organisational models describe desired and undesirable situations, in practice agents need to decide on a concrete plan of action. This creates a distinct gap between the representation of an (abstract) organisational norm and the (concrete) plans of an agent.

Traditionally, the very notion of agent autonomy refers to the ability of individual agents to determine their own actions, plans and beliefs and, according to [Wooldridge and Jennings, 1995], to the capability of an agent to act independently, exhibiting control over its own internal state. Nevertheless, it is not always clear how norms affect an agent's autonomy and most systems fail to capture complex normative reasoning able to, for example, resolve conflicts between various norms at runtime.

Traditionally, possible deviations from desired behaviour are coped with by operating through a single utility function or by applying hard constraints to the agents. Nevertheless, the former results as rather inflexible while the latter might lead to exponentially large numbers of decision paths and, most of the times, such a hard coded modelling leads to rigid and complex agent specifications. For example, while it is possible for a firefighter agent to allow a restriction "*whenever there is a fire, call reinforcements before entering the building*", it may not always be the desired thing to do and under some specific circumstances (if, for example, the nearest fire station is far away, implying that the reinforcements might take too long to arrive) one might desire to deviate from such a restricting measure.

Further to this, it is not always possible to apply constraints in a dynamic non-deterministic environment where factors that can modify/influence the environment may exist. Generally speaking, agents living in environments of this kind require a certain level of sophistication and autonomy to defend their affairs and this can hardly be done via constraint mechanisms. Embedding normative restrictions hard-coded into an agent is not a choice, as the normative restrictions of a virtual society are not always known beforehand to the agent's developers.

1.2 Aim of the Thesis

While human societies offer a great deal of inspiration for the development of norm-based artificial societies and implementation architectures, few existing frameworks provide a functional implementation of normative concepts and therefore flexibility in terms of norm adoption and reasoning about courses of action.

Within complex social setups, the main advantage of normative specifications over other governing mechanisms is that norms provide an explicit description of social expectations without giving precise information on how the agents are supposed to bring these about and, allowing at the same time the agents enough flexibility and some level of autonomy to react to different states of the system.

From the agent's perspective, it must be its own capabilities and aims that determine the specific way and the reasons for which an agent will enact its role(s), and the behaviour of individual agents is driven and guided by their own goals and abilities [Dastani et al., 2003]. That is, agents bring their own ways into the society as well, but need to be able to check whether their actions are in accordance to organisational specifications.

In this thesis we tackle and provide a practical solution to the following question:

"*How to model an autonomous, goal-driven agent that is able to take the environment's normative influence into account in its decision making?*"

Our thesis aims to answer the question posed by providing a generic mechanism to support practical normative reasoning[2] in environments where norms are enforced, not regimented [Jones and Sergot, 1993]. In this work we focus on the regulative aspect of norms, seen as a way to model the governance of distributed, agent-oriented systems. Thus, norms may refer to explicit behavioural commitments, restrictions or other types of impositions that the agent is expected to comply with.

We envision normative agents that dynamically interpret, apply and reason over norms and decide whether to comply with the enforced norms or alternatively deviate

[2]From now on, we will use the term practical normative reasoning to refer to a norm-influenced version of practical reasoning (a process for deciding what to do [Bratman, 1987]).

from predetermined behaviour. This way, agents become norm-autonomous while operating in dynamically changing and possibly shared by other agents environments, normally represented by non-static, complex domains.

In contrast to traditional methods where agents use pre-factored plans, we consider the dynamic, real time[3] creation and evaluation of plans for achieving specific goals as the main means-ends reasoning mechanism. In addition, when being norm-aware, this mechanism must be able to generate the most profitable trajectory towards the agent's goal(s) while at the same time respecting the environment's normative influence. When new information occurs, the reasoning mechanism should be able to drop current processes and regenerate plans, taking the new information into account.

Functioning in multi-agent societies, the agents should dynamically become aware of new norms. Consequently, these newly adopted normative standards must influence the agents' practical reasoning. The agents must also be able to invent strategies that handle conflicting situations that might occur while operating in a complex normative context.

Finally, preferences play a crucial role in guiding our choices among possible outcomes and actions throughout a decision making process. They normally represent the most satisfactory or most plausible states when expressed in a given context and often taken into consideration in order to select optimal choices and avoid risk-prone situations. Given the importance of preferences in modelling real-world scenarios, the agents must be endowed with the ability to take users' preferences and other weighing factors into consideration, performing in this way a fully qualitative decision making.

1.3 Thesis Objectives

This thesis presents a framework for practical reasoning within normative systems. A formalisation of a normative model is provided and an approach towards reasoning within the model involving a standard action-based planner is explored. The main objectives of this thesis, which will advance the current state of the art, are:

- Provide formal semantics of an organisational normative model within which an agent operates.
- Specify a conceptual metamodel abstracting the representation of the reasoning mechanism elements such as norms, actions, current state of affairs, goals and action costs.
- Using notions of the planning research area, establish a formal connection between the reasoning elements (norms, actions, current state of affairs, goals and action costs) and the normative model and investigate ways in which a course

[3]As we will see in Subsection 3.1.2 we will use the term *real time* to refer to *soft real time* performance, that is, non strict computational periods of time within an acceptable range. We consider the definition of those acceptable ranges to be domain-dependent.

of action satisfying the desires of the agent can be produced, while taking the norms into consideration. This process, from now on referred to as *normative planning*, includes the specification of plans which not only comply with the agent's goals but do so through a minimal cost procedure.

- Based on the previous, implement a norm-aware planner.
- Implement a prototype normative agent:
  - Design a general architecture of a BDI reasoning agent which incorporates a normative reasoning mechanism into its practical reasoning to allow a norm-aware autonomous decision making process.
  - Create a mapping between the conceptual metamodel that describes the reasoning mechanism to the inputs of the prototype. This mapping will result in the automatic production of the inputs for the reasoner implementation at runtime, easing integration.
  - Wrap up the planner and incorporate it within a BDI agent cycle.
- Validate the prototype architecture through a life-like use case.

1.4 Out of Scope Issues

While the following are related topics that could be of interest, they are not examined within the scope of this thesis:

- Constitutive aspect of norms. Although norms can be categorised in two main types, *regulative* (regulating pre-existing forms of behaviour) and *constitutive* (specifications that state what counts as what) [Searle, 1969], in this thesis we focus on the regulative aspect of norms. Both types are discussed in Section 2.2.2.2.1 and some frameworks dealing with them can be found in Section 2.2.2.4.1.
- Complex legal notions such as *empowerment* and *delegation* are extensively mentioned in the scientific literature [Jones et al., 2013; Dignum, 2004]. In particular, empowerment, seen as the normative authority that one might have to act in a specific way or bring about a specific situation, is explicitly separated from the permission to do something that one might have. As we will see in Section 2.2.2.2.2, where a research background in empowerment is explained, the notion is generally modelled through constitutive norms. However, in our thesis we do not deal with the notion of power, as the main focus is the regulative aspect of norms.
- Verification of norms. We assume that agents operate within an environment where norms are correctly expressed and consistent. Although when agents are operating in normative societies it is necessary to provide mechanisms that perform multi-level verification (detecting and/or correcting possible conflicts) of norms through some kind of logical reasoning, we do not intend to examine

this aspect in our thesis. Relevant work focusing on the verification of norms can be found in Section 2.2.2.3.4.

- Monitoring of norms. An agent will need to check the normative status of all norms applying, with a mechanism close to (institutional) norm monitoring approaches. However we are not interested in the techniques needed to observe the behaviour of large agent societies. Relevant work concentrating on this domain is described in Section 2.2.2.3.5.
- Multi-agent planning. While ideally it is desirable for agents to be able to operate in a society comprising of multiple agents, this thesis reviews the existence of a single agent and therefore concentrates on the single-agent planning area while intentionally leaving out any multi-agent planning methodologies and resource negotiation for future work.
- Norm creation and adoption. Agents joining a normative environment for the purpose of executing specific collaborative tasks usually have to adopt or choose to ignore norms representing certain rules and regulations. Adoption of norms can cause problems – an agent maybe already hold norms that would be in conflict or inconsistent with new norms it adopts. In this case, the answer to the question "*is the set of norms declared in the context consistent with the set of norms the agent currently holds?*" is not simple in a real-world situation. In this thesis we take for granted that norms pre-exist and are imposed by the organisation. While it might not always be the case in real-world scenarios, we simplify and isolate our research by assuming that norms have already been created and that the agent adopts by default the full set of them as we do not intend to explore these two issues. Research work relevant to this topic can be found in [Vasconcelos et al., 2007; Kollingbaum and Norman, 2003]. Norm adoption is also briefly discussed in Section 2.2.2.3.1.

1.5 Claims

Based on our thesis objectives, we take a step forward assuming some reasonable claims made when dealing with practical reasoning with norms.

C.1: Throughout the agent's deliberation process, means-ends reasoning can be performed by invoking a planner instead of using pre-compiled plans. This allows the agent to have a more flexible and pro-active behaviour towards achieving its goals and the capability of on-demand replanning.

C.2: The planning process can be influenced with norms. While this is not be directly offered by current planner implementations, we can represent normative influence by enriching the planning domain and problem with the appropriate normative restrictions (represented as additional actions, goals and other features allowed by different planners).

C.3: Standard Deontic Logic is not sufficient to capture practical, complex aspects of norms' functionality. It does not provide operational semantics (i.e. norm instances) that can be directly translated by computational systems.

C.4: Temporal logic may be used to capture the norms' lifecycle. Aldewereld et al. have already introduced mechanisms to enforce normative behaviour through temporal expressions [Aldewereld et al., 2006]. We take it one step further and assume that temporal logic expressions can be used to capture complex operational norm lifecycles, supporting norm instances and allowing norms to be possibly violated. These expressions might be directly introduced into planners that take into account such temporal rules or indirectly represented in planners that do not support such temporal rules, in order to produce norm-consistent plans.

C.5: An existing agent framework can be extended with norms. Instead of building an agent framework from scratch, we can choose an existing one and extend its language to incorporate norms so that the agent is aware of their existence. Our normative reasoning mechanism can be integrated within the agent framework, (possibly partially) replacing its existing reasoning engine.

C.6: Actions with costs can be used for modelling domain and user preferences. By assigning (possibly conditional) cost functions to the actions available to the agent we can model the weight each action has on various pre-determined agent factors when performed. Having a planner sensitive to the costs information, the objective will be to calculate near-optimal plans with respect to some pre-determined function that represents the quality of the plan for the agent.

C.7: A norm might be defined on an abstract level, allowing a flexible representation like a template for the construction of concrete instances. Throughout time, many instances of a particular norm might occur. In this thesis we adopt the view that norm compliance depends on the compliance of its instances occurring through time. We provide formal notions for norm compliance in Chapters 4 and 5, according to the two different norm formalisations and put them to use when designing and implementing the normative reasoner in both approaches.

The validity of all these claims will be explored through Chapters 3, 4 and 5, and will be summarised in Chapter 6.

1.6 Structure of the Thesis

The remainder of the thesis is structured in the five following chapters:

Chapter 2: This chapter is dedicated to a state of the art analysis, covering work on normative environments, and a background on reasoning frameworks. Works using a more practical approach used in distributed systems and concerning a particular type of normative environments (e.g. contractual environments) are also discussed. The

background on action and planning languages, which are closely related to our thesis, is also detailed.

Chapter 3: In this chapter we make an extensive requirements analysis of our framework that will support normative reasoning. The decisions that derive from the analysis are thoroughly discussed. We then provide the baseline that supports our proposal given in Chapters 4 and 5. The conceptual framework, based on the requirements presented, and the formalisation of the normative elements involved through metamodels is then displayed. The metamodels are the foundation of the elements and their usefulness lies in the fact that they provide both a conceptual abstraction and some technology/platform independence. We additionally depict the metamodels of the elements that support the framework in a graphical way. After, our system architecture, which interconnects the conceptual framework elements, is presented and explained. Finally, we discuss some basic issues occurring when dealing with norm formalisation and representation.

Chapter 4: This chapter bridges a semantic gap between the vast theoretical research and few practical implementations done on normative systems[4]. We do this by defining a lifecycle of norms and by reducing deontic statements to temporal logic expressions. At the same time, we explore several deontic properties of subreductions, showing the correlation with Standard Deontic Logic. We further show how the semantics defined in temporal logic can be translated to planning control rules, for practical normative reasoning. We show the feasibility of the translation of these semantics to actual implementation languages (TLPLAN) and present experimental results over a real-life use case.

Chapter 5: We introduce a proposal extending the one in Chapter 4 in order to overcome the barriers encountered in the initial approach. Based on the formal model introduced in Chapter 4, the semantics of norms are extended to include multiple layers of repair norms. We define the new norm lifecycle with rules and we implement a normative reasoner by applying those rules within a planning system, integrate it into an existing BDI agent framework, 2APL, and provide experimental results which show the usefulness and efficiency of the approach.

Chapter 6: We provide a summary of the work in the thesis and its contributions, discuss the limitations of our approach and then outline some of the directions for future work and extensions.

Appendix A: In this appendix we present our previous work on formalising normative environments, which inspired the operational semantics describing the norm lifecycle and semantics for its interpretation. These include norm activation, discharge, fulfilment and violation.

[4]The formalisation presented in this chapter is a collaborative work done together with Sergio Álvarez-Napagao which has been presented in [Panagiotidi et al., 2013]. It has been based on our efforts to bring together two main areas, institutional-level norm monitoring of normative systems and individual agent norm-aware reasoning, on a semantic as well as on a practical level.

Appendix B: We provide here the details of the use case implementation within our normative framework. The code for the normative planning as well as the code for the 2APL agent for the same example are presented.

Appendix C: In this appendix we make the proofs of the deontic logic reductions and the properties shown in Chapter 4 available.

Chapter 2

Related Work

In Chapter 1 we motivated our interest in norm-governed agent systems and in particular in the need for autonomy. To make such systems more autonomous, a form of intelligent normative reasoning is needed. This means that an agent should be able to make personal informed decisions while at the same time taking into account normative influence of the environment, that being done by weighing personal and domain factors against the norm compliance or non-compliance consequences.

In order to better comprehend the work we are presenting in this thesis, it is necessary to provide some details on the state of the art. In this section we detail work that has been done in the fields of agent reasoning, normative systems and contracting systems (seen as a particular type of normative systems). Research relevant to reasoning and planning (such as action languages to express operational semantics of actions) is also provided.

2.1 Agent Orientation

As the computing industry transfers its focus from individual, single-machine systems towards distributed, dynamic and powerful systems, more challenges are brought into the picture. According to van Steen [van Steen et al., 2012], as connectivity increases in such large-scale systems, distribution transparency is needed in order to disguise the processes, data exchange and control in them. With all the unpredictability that is brought by the spontaneous user joining in modern distributed systems, automated processes should be able to automatically handle human inputs. Heterogeneity also needs to be addressed and collaborative systems that effectively capture and analyse users' behaviour and intercommunications need to be devised. At the same time such systems should be self-configuring and able to secure their correctness, manage their stability by handling faulty or defective components and maximising their own performance [van Steen et al., 2012].

In such large-scale distributed systems, agent-based systems [Wooldridge and Jennings, 1995; Weiss, 1999] might provide answers. Presently, agents are being used in a

large number of science and technology applications from small ones such as online shopping agents to bigger ones such as traffic and transportation systems.

Software agents acquire some knowledge about the world in which they operate, so that they can deal with most of the minor issues they come across in operation by themselves, without any intervention of the user (except in concrete justified occasions). One of the most cited definitions of an agent is the following:

"*An agent is an encapsulated computer system that is situated in some environment and that is capable of flexible, autonomous action in that environment in order to meet its design objectives.*" [Wooldridge and Jennings, 1995]

Autonomy refers to the agent's ability to exist and manifest proactive behaviour and possibly socially interact and collaborate with other agents within a specified environment, while at the same time perceiving and also capable of interacting with the environment itself. Autonomous acting implies that the agent possesses cognitive and decision making abilities while the objectives play an important role in indicating the way the agent will use these abilities in order to achieve them.

Many processes in the world can be conceptualised using the agent metaphor. Nevertheless, the metaphor alone is not enough as at times the number of agents may be too numerous to deal with them individually. The agent is socialised in a "field", an evolving set of roles and relationships in a social domain, where various resources are at stake. It is then more convenient to deal with agents collectively as agent societies. The result of such a conceptualisation is a *multi-agent* (or *social*) description.

2.1.1 Practical Reasoning in Agents

In general, practical reasoning is directed towards action - a decision process for deciding what to do [Bratman, 1987; Wooldridge, 2001]. It involves two essential activities:

a) *deliberation* - deciding WHAT goals to achieve and
b) *means-ends reasoning* - deciding HOW to achieve these goals.

Deliberation is concerned with determining *what* one wants to achieve (considering preferences, choosing goals, etc.) and generates or modifies our intentions (which serve as the interface between deliberation and means-ends reasoning) [Bratman, 1987]. Means-ends reasoning on the other hand is used to determine *how* the goals (objectives) are to be achieved. That includes thinking about suitable actions, resources and how to "organise" activity and generating plans which are turned into action sequences.

Both Wallace [Wallace, 2009] and Bratman [Bratman, 1987] offer similar interpretations for practical reasoning. Specifically, Bratman states:

"Practical reasoning is a matter of weighing conflicting considerations for and against competing options, where the relevant considerations are provided by what the agent desires/values/-cares about and what the agent believes." [Bratman, 1987]

Several characteristics of practical reasoning can be summarised in the following points:

- Objectives can create obstacles for agents, who need to come up with ways for reaching them.
- Objectives provide a "filter" for adopting other objectives, which must not conflict.
- Agents follow the achievement of their objectives, and are disposed to make more efforts if their attempts are unsuccessful.
- Agents believe their objectives are possible.
- Agents do not believe they will not bring about their objectives.
- Under some conditions, agents believe they will accomplish their objectives.
- Agents do not need to intend all the presumed side effects of their objectives.

2.1.1.1 The BDI model

As a consequence of the above, an approach for the development of reasoning agents is to describe them as an "intentional system" [Dennett, 1989] in terms of mental attitudes such as "beliefs", "desires" and "intentions" (BDI). In recent years the BDI architecture [Bratman, 1987] has become a de facto standard for agent models and is generic enough to enable the modelling of both natural and artificial agents. The BDI model is inspired by efforts to understand mental attitudes and simulate practical reasoning in humans (e.g. [Bratman et al., 1991; Cohen and Levesque, 1990]. The BDI reasoning cycle consists of two important processes: *deliberation* (determining what goals to achieve) and *means-ends analysis* (determining the way to achieve these goals). The reasoning process analyses and decides which beliefs and desires to pursue, discarding the ones that are unsuitable at the current time and the chosen options then become the agent's intentions. Typically, the agent will sort the set of 'suitable' intentions based on some evaluation function and choose the one with the highest score. Intentions are a crucial element in the reasoning process as they can be achieved through actions, so they determine the agent's behaviour. Agents must believe that they can satisfy their intentions. Important aspects of intentions are [Bratman, 1987; Wooldridge et al., 2000]:

- They conduct the means-ends reasoning process: after an intention is determined, trying to achieve it involves deciding in which way this is possible.
- They constrict the agent's future deliberation: options that are not consistent with its intentions will not be considered. Thus, the agent is assumed to avoid states in which it would have two contradictory intentions.

- Persistency: agents will normally stick to their intentions until they are achieved unless they are discovered to be unreachable. When an intention is considered impossible to achieve as well as how to balance low-commitment (the tendency to give up intentions too easy) and over-commitment (the tendency to excessively stick to intentions) in agents are still open issues that will typically depend on the domain and the problem the MAS is tackling.
- Influence beliefs: plans are supported and formed by the knowledge that the intentions will be pursued and achieved.

In short, agents have a set of beliefs that can be seen as their internal state. Beliefs reflect the agent's awareness of the environment, forming this way its unique, internal environment perception. Through an internal filtering mechanism over beliefs and intentions that considers the actual agent and environment state, the agent determines its options (desires). Intentions are decided through a deliberation over its beliefs, desires and intentions and they form a central component of the agent. They represent the states the agent is determined to bring about and towards the achievement of which the agent will invent and follow a plan of action.

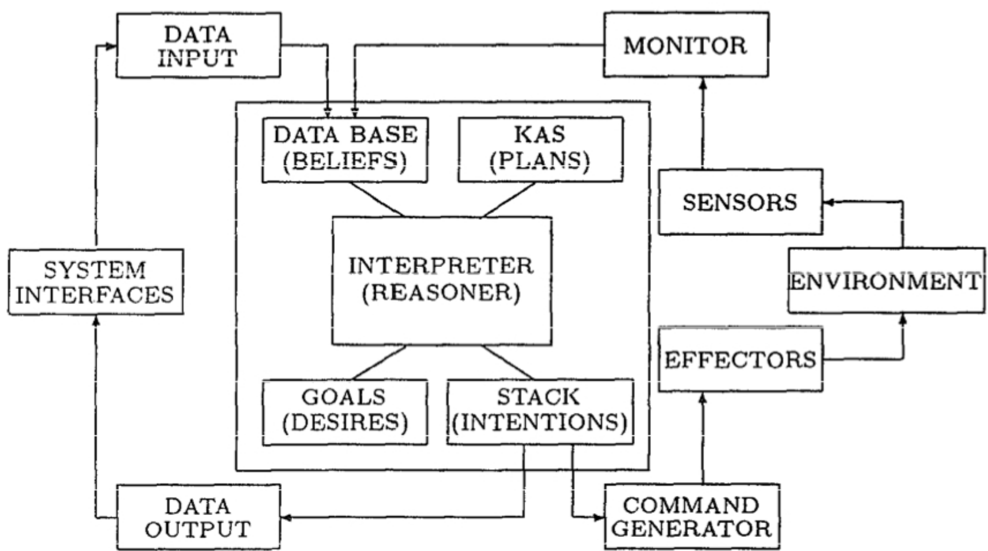


FIGURE 2.1: The PRS agent model (source: [Georgeff and Lansky, 1987])

There exists a rich background in formalising BDI models [Rao and Georgeff, 1995]. BDI agent models have also been extensively used in order to build solutions to problems in several different applied areas such as spacecraft management, telecommunications and air traffic control. One of the earlier implementations is the PRS system [Georgeff and Lansky, 1987]. PRS (Figure 2.1) is an implementation of a BDI architecture which maintains data structures for beliefs, desires and intentions. It adopts a reactive planning approach and maintains a library of pre-constructed plan procedures or "Knowledge Areas". These pre-constructed plan procedures represent conditional sequences of actions that can be executed to reach specific goals or to directly respond to specific situations. PRS follows a partial planning methodology where the agent

creates a partial "hierarchical plan" by specifying a method to reach a goal as a plan procedure and following this procedure. The plan procedure can be of raised complexity and one of its aims is to form sub-goals. The agent must then choose suitable means for reaching those sub-goals. In that way, PRS expands plans incrementally and can therefore react to non-static situations and modified goals by altering its course of action.

2.1.1.2 Practical Frameworks Implementing BDI agents

Modern implementations of the BDI agent architecture include GOAL [Hindriks, 2009], Jason [Bordini and Hübner, 2006], JACK [Winikoff, 2005], Jadex [Braubach et al., 2005], 2APL [Dastani, 2008] and 2OPL [Dastani et al., 2009a]. All the frameworks share several features, common in the BDI agent architecture and they all (with the exception of Jadex which uses a hybrid approach) provide an agent-oriented language in which the agent's reasoning elements are defined. Further to this, they implement a basic execution (reasoning) cycle, through which they apply rules and update their knowledge bases. Below we present the details and particularities of each one of the frameworks.

Goal-Oriented Agent Language (GOAL) [Hindriks, 2009] is a high-level language to design BDI agents. It provides programming constructs to represent beliefs and goals in a declarative manner (the term *declarative goals* is used to indicate goals that specify a state or set of states of the environment that the agent wants to achieve, rather than actions or procedures on how to achieve such states). GOAL supports various types of actions such as user defined actions, built-in actions, and communication actions. It also includes action selection rules to support the action selection mechanism by providing action selection strategies. A particularity of GOAL is the absence of plans. In addition to the work described, Hindriks et al. research on how goals that can be expressed in temporal logic, called *temporally extended goals* and how these can be managed within an agent's reasoning process [Hindriks et al., 2009]. They use GOAL as basis for the semantics and the elements they use. They distinguish between two types of goals, *achievement goals* and *maintenance goals* and focus on the "*bounded*" goals, meaning goals that have a finite time horizon to be achieved, in contrast to "*unbounded*" that can have infinite horizon to be achieved. In their semantics they allow the expression of goals and beliefs in terms of LTL formulas and they define the mental state of the agent in terms of beliefs and goals of the agent. An agent is able to update through a *progression* process (i.e. a computational process) its belief and goal base after the execution of some action. In possession of a set of actions and a set of *action rules* specifying under what conditions to perform an action, they perform reasoning over which actions to select and execute based on the action rules. Additionally, they develop a method to select those actions that will result in minimum violations of goals (that is, goals not accomplished until the maximum lookahead horizon).

Jason on the other hand [Bordini and Hübner, 2006] interprets and implements the operational semantics of an extension of AgentSpeak (previously AgentSpeak(L)) [Rao, 1996], a logic-based programming language inspired by the PRS model (see Section 2.1.1.1) for building rational agents, developed by Rao. Jason is a platform for the development of multi-agent systems, with many additional features. The performatives that are currently available for agent communication in Jason are largely inspired by the Knowledge Query and Manipulation Language (KQML)[1] [Finin et al., 1995]. Additionally, the conduct of the agent within the environment is specified in AgentSpeak(L), a restricted first-order logic language supporting events and actions. The language includes a *set of plans* which compose its *plan library*. The current belief state of the agent (seen as the current state of the agent, that is, the state of itself, the environment and other agents) is a set of *beliefs* and initially it comprises a set of *base beliefs*. The states that the agent, influenced by internal or external situations, wishes to bring about can be seen as its desires. The AgentSpeak interpreter also manages a set of *triggering events* (changes in the agent's beliefs or goals) and a set of *intentions* which are the adopted plans to react to such stimuli. Plans consist of a head (containing a triggering event - the addition or deletion of beliefs or goals - which initiates the execution of the plan) and a body (containing a sequence of goals that should be achieved or tested and actions that should be executed). Whenever a triggering event occurs, AgentSpeak reacts to it by executing the plan that contain it in its head condition. As the creators claim, "*[...] this shift in perspective of taking a simple specification language as the execution model of an agent and then ascribing the mental attitudes of beliefs, desires, and intentions, from an external viewpoint is likely to have a better chance of unifying theory and practice*" [Rao, 1996].

Jadex [Braubach et al., 2005] is a BDI-inspired reasoning engine that allows for programming intelligent software agents in XML and Java. The reasoning engine can be used on top of different middleware infrastructures such as JADE [Bellifemine et al., 1999]. Jadex supports a practical reasoning cycle including goal deliberation as well as means-ends reasoning. The first is responsible for deciding which of the existing goals are currently pursued and the latter has the task to find means for realising a specific goal by applying suitable plans. Unlike other BDI agent systems (e.g. Jason) where beliefs are represented in some kind of first-order predicate logic or using relational models, in Jadex a hybrid language approach is used, where declarative agent information is separated from procedural plan knowledge. In the XML based agent definition file (ADF) the beliefs, goals and plans (static specifications) of an agent type are defined, whereas Java classes are used for encoding the plan bodies (dynamic behaviour). An object-oriented representation of beliefs is used, where random objects

[1]KQML is a language designed to support interaction among agents. It contains information concerning various levels of communication, such as the parties involved, the actual content exchanged and the language in which the content is expressed. The *performatives*, being the core of KQML, define the permissible interactions which agents may use. Performatives contain arguments which specify the protocol used to exchange messages and the *speech act* that the sender attaches to the content of the message and optionally the sender and receiver. Performatives can be a query, command, assertion or any other other speech acts agreed upon.

can be represented as named facts (beliefs) or named sets of facts (belief sets). Further to this, goals are represented as explicit objects contained in a goalbase, which is accessible to the reasoning component as well as to plans if they need to know or want to change the current goals of the agent. Four different goal types that refine the basic goal lifecycle in different ways are distinguished (perform, achieve, maintain and query goals). The framework does not expect all adopted goals to be consistent with each other, as long as only consistent subsets of these are aimed at, at any point in time. The plan head specification is similar to other BDI systems and mainly specifies the circumstances under which a plan may be selected, e.g. by defining preconditions for the execution of the plan and events or goals managed by it. A special feature of Jadex is that it can support both sequential and parallel execution of plans and it can also allow for a plan execution to wait for (pause and resume after) the execution of an action or the reception of some message.

JACK Intelligent Agents [Winikoff, 2005] is an agent framework developed on ideas of other reactive planning systems and can be, in this respect, considered quite similar to Jason, 2APL and Jadex. JACK is an extension of Java, based on logic, with a number of syntactic constructs allowing to create, select and execute plans and belief bases in a graphical manner. In JACK, like in Jason, goals are represented a as special type of event (goal-event). The way the agents are implemented in both frameworks is that the agent is not aware of the pursuing goals, but instead executes a plan as a response to the occurrence of an event. JACK, like Jadex, lacks formal semantics of beliefs and goals, however, being a commercial platform it provides many supporting tools and is being used extensively in industrial applications.

2APL [Dastani, 2008] (preceded by 3APL) is a Prolog-based agent programming language based on the BDI model. Briefly it is composed of beliefs, goals, belief updates (actions), plans, events and three different types of procedural rules. The environments in 2APL are implemented as Java objects. Beliefs and goals are expressed in a declarative way as Prolog facts and form, respectively, *belief* and *goal bases*. Any Prolog construction can be used in the belief base. *Belief updates* update the belief base of an agent when executed. Such an action is specified in terms of preconditions and postconditions (effects). An agent can execute a belief update action if the precondition of the action is derivable from its belief base and its execution modifies the belief base so that the effect of the action is derivable from the belief base after. *Procedural rules* are complex Prolog statements that might instantiate plans for the agent to execute. These serve as practical reasoning rules that can be used to implement the creation of plans during an agent's execution. In particular, they consist of: *planning goal rules*, *procedure call rules*, and *plan repair rules*. The first type generates plans for reaching goals, the second type handles (internal and external) events and received messages, and the third type handles and repairs plans that might fail. A *plan* is a program to be executed. It might consist of belief updates as well as *external actions*, i.e. actions that affect the environment. According to the creators, the use of declarative goals (in

contrast, for example, to implicit goals originating from the execution of plans triggered by events as in, for example, Jason) in the language adds flexibility in handling failures. If following a plan does not end up reaching its corresponding declarative goal, then the goal remains in the goal base and can be used to trigger a different plan. An distinguishing feature of 2APL, like Jadex, is that it provides a programming module to implement non-interleaving execution of plans.

2OPL (Organisation Oriented Programming Language) [Dastani et al., 2009a] sees a multi-agent system as a system where "agents' behaviours are regulated by an organisational artefact". In 2OPL a logical representation structure to keep the organisational specification apart from an environment is used. The language allows the programmer to model the organisation under four sections namely *Facts*, *Effects*, *Counts-As Rules* and *Sanction Rules*. In addition, actions have constraints and can cause violations and be sanctioned. Violations can be handled in two ways. Regimentation[2] [Jones and Sergot, 1993] is making the violation of norms impossible for agents. It means blocking the action that causes a regimented violation completely. Enforcement [Jones and Sergot, 1993] is allowing the violation of norms first and then sanctioning the actors of the violation. An interpreter for the 2OPL language is explained in [Adal, 2010].

2.1.2 Agent Planning

Agent planning is a broad field of AI with a strong background of research focusing on formalising models and algorithms as well as many practical applications. In principle, the agent community acknowledges that in ideal agent implementations the agents should be able to dynamically generate plans at execution time. However, existing planning implementations were unable to provide plans at real-time, and thus the agent community and the planning community have remained separate and, in general, little effort has been made to use planning when implementing an agent's decision making process. For some researchers, planning is a sort of automatic programming. The planner takes a symbolic description of the world, the target (goal) state and a set of agent capabilities (actions) and attempts to find a sequence of actions that achieve the target.

Formally, the classical planning problem has been defined by Weld [Weld, 1999] as follows. Given:

- the known part of the initial state of the world (in a formal language, usually propositional logic),
- a goal (that is, a set of goal states), and
- the available (atomic) actions that can be performed, modelled as state transition functions,

[2]Further discussion on norm regimentation is given in Section 2.2.2.2.

the task is to compute a plan, i.e. a sequence of actions that transforms each of the states fitting the initial configuration of the world into one of the goal states.

In recent years *metric planning* has gained attention in the planning research community. Metric planning can be seen as the numerical extension of the planning problem, where actions modify the value of numeric (or possibly of other type) state variables. This is seen as necessary since most domain problems consist of various real-world elements that need to be represented by quantified variables, called *fluents*. The task in the case of the metric planning is to determine a plan that achieves the goal criteria and at the same time optimises an objective function (consisting of the aforementioned variables).

In this section we discuss the different action formalisms invented for agent planning and several implemented planning frameworks.

2.1.2.1 Action Language Formalisms

Action languages are formal ways of representing the human notion about actions and their effects. A basic element common in most action languages is the *transition system*. Action languages model actions and world situations in such a way that these can directly or indirectly be mapped to a transition system, where actions form transitions between states.

Situation calculus [McCarthy and Hayes, 1969; Levesque et al., 1998] is the oldest and most widely used logical formalism for describing dynamic domains. It was introduced in 1963 by McCarthy and subsequently refined. Situation calculus is a second-order framework for representing dynamically changing worlds in classical first-order language. Variables might be of different types including *objects*. Situation calculus represents the world and its change as sequence of *situations*, where each situation is a term that represents a state and is obtained by executing an action ($action \times situation \rightarrow situation$). *Action* functions are a main element of the formalism and a special binary predicate $Poss : action \times situation$ is used to indicate under which situation s an action a is executable. Additionally, *effect axioms* describe changes in situations that result from the execution of actions. Situation calculus allows for the representation of the *initial situation* s_0 and a special function symbol $DO(a,s)$ which describes the situation obtained after performing action a at situation s. Finally, two types of *fluents*, *relational* fluents, of type $(action \bigcup object)^n \times situation$ and *situational* fluents of type $(action \bigcup object)^n \times situation \rightarrow action \bigcup object$ are considered as properties of the world model change. The main problem that occurs when modelling in situation calculus is the so-called *framing problem* (or *frame problem*). That is, since there is no way to indicate the properties that remain unchanged within a domain after the execution of an action (the non-effects of actions), there occurs a need to represent a large number of frame axioms, leading to a great complexity of the domain. The

successor state axioms, one for each fluent, tackle the problem, by making sure that effect axioms fully specify all the ways in which the value of a particular fluent can be changed as a result of performing an action.

Event calculus [Kowalski and Sergot, 1986], introduced by Kowalski and Sergot, is another approach for encoding actions into first-order predicate logic. This calculus was designed to allow reasoning over time intervals and includes the non-monotonic inference *negation as failure*. The main difference between event calculus and situation calculus is conceptual: event calculus is based on points in time rather than on global situations and fluents hold at points in time rather than at situations, as it happens in situation calculus. The events, their effects and durations are expressed in Horn logic [Horn, 1951]. A special predicate $HoldsAt(f,t)$ is used to indicate that a fluent f holds at a given time point t. The predicate $Initially(f)$ indicates that a fluent f holds at time 0 and $t_1 < t_2$ that time point t_1 is before t_2. The effects of actions are given using the predicates $Initiates(a,f,t)$ and $Terminates(a,f,t)$ where t indicates a specific point in time. Predicate $Happens(a,t)$ indicates that an action a takes place at time t. Finally, predicate $Clipped(t_1,f,t_2)$ indicates that fluent f has been made false between time points t_1 and t_2. The frame problem can be solved in a similar fashion to the one used in situation calculus, by adding axioms stating that a fluent is true at some time point if it has been made true before and has not been falsified in the meantime as well as stating the opposite case in which a fluent is false at a given time. According to [Eshghi, 1988], planning in the event calculus can be considered as an abductive task, where a logic programming technique is applied to generate plans using representations of *Initiates*, *Terminates* and *Happens* predicates. While there are some earlier implementations of such abductive event calculus planners [Jung et al., 1996; Shanahan, 2000], these are rather inefficient, due to the nature of Prolog employed to act as the abductive theorem prover.

The fluent calculus [Thielscher, 1999, 2005] is yet another formalism for expressing dynamical domains in first-order logic. It is a variant of the situation calculus; the main difference is that situations are considered representations of states. The world can be in the same state in different situations, but the state in every situation is unique. The main contribution of the fluent calculus is its solution of the inferential frame problem by the means of state update axioms. In [Thielscher, 2005] a fluent calculus executor (FLUX) is presented.

For domains of incomplete and/or inconsistent information one can find action specification languages such as $\mathscr{A}$ [Gelfond and Lifschitz, 1993, 1998] and $\mathscr{K}$ [Eiter et al., 2003], which allow modelling dynamic domains with incomplete and inconsistent information. Both languages are close in spirit to answer set semantics (Answer Set Programming or ASP [Dimopoulos et al., 1997; Marek and Truszczynski, 1999]). $\mathscr{A}$ allows the effects of an action to be conditional. It roughly contains: a) *effect propositions* of the type: a `causes` f `if` $p_1 \ldots, p_n, \neg q_1, \ldots, \neg q_r$, b) *value propositions* of the type: f `after` $a_1, \ldots, a_n$ and 3) propositions stating the initial situation of the system of the type: `initially` f. Due to its ASP orientation, language $\mathscr{A}$ is directly translatable,

can be read by and produce plans via an ASP solver. Language $\mathscr{K}$ has similar syntax and semantics to $\mathscr{A}$. A system that uses $\mathscr{K}$ is the $\texttt{DLV}^K$ declarative, logic programming based planning system. $\texttt{DLV}^K$ allows for parallel execution of actions and both strong (¬) and weak (`not`) negation. The main strength of the language $\mathscr{K}$ is that it can deal with cost functions in the action descriptions and that it can reason about incomplete states. On the other hand, it lacks the ability to express conditional effects, durative actions and flexible numeric handling.

Several other frameworks are also concerned with the design of action descriptions. In [Baral et al., 1997; Baral, 2003] Baral et al. detail an action description language to express causal laws over the effects of actions and the values of fluents in possible states of the world. Additional work of some of the authors mentioned previously focuses on different aspects of reasoning over dynamic domains [Baral and Gelfond, 1999] and over properties of actions [Gelfond and Lifschitz, 1993].

The language $C+$ [Giunchiglia et al., 2004] is a formalisation for defining and reasoning about the effects of actions and the persistence (inertia) of facts over time. An action description in $C+$ is a set of $C+$ laws (containing *static laws* of type "'**caused** *F* **if** *G*" and *dynamic laws* of type "**caused** *F* **if** *G* **after** *H*") which define a labelled transition system with explicit semantics. The main problem of the original language $C+$ is that it is purely propositional, not allowing variables. An implementation of a variation of $C+$ in Prolog supporting a range of querying and planning tasks is available, notably in the form of the "Causal" Calculator CCalc [McCain, 1997].

An important disadvantage of all the aforementioned languages is that their semantics does not deal with conditional effects and numeric functions and costs (with the exception of $\mathscr{K}$) in the domain. Additionally, as already explained, while there are some practical implementations of these frameworks, they are limited by the execution times of Prolog or ASP solvers and there do not exist sufficiently efficient implementations that can compete with current classical planners and hierarchical planners. We discuss the latter two in the following section.

2.1.2.2 Practical Frameworks Implementing Action Languages

Two basic planning areas that have been widely explored in the last decades are *classical (deterministic) planning* and planning with *hierarchical task networks*. In this section we discuss several frameworks implementing action languages in both areas. As we will see later in this section, the newest approaches intend to support *control rules* over the calculated paths on top of everything else.

In classical planning, actions that correspond to deterministic state transitions are used to define fully observable and deterministic domains and the objective is to achieve a set of goals. When referring to classical planning, the Planning Domain Definition Language (PDDL) [Ghallab et al., 1998; Fox and Long, 2009], an extension of STRIPS

[Fikes and Nilsson, 1972], is the most commonly used language for formulating deterministic classical planning domains and problems. It started off as a pure propositional framework but got extended in various directions (durative actions, metric planning, probabilistic planning, etc.) and consequent versions that accommodate complex domains have risen within the last ten years, allowing for sharing domain models as well as representing domains in a commonly accepted and comprehensible manner. While STRIPS (and as a consequence PDDL too) can be considered more restricted in expressional power compared to situation calculus (described in Section 2.1.2.1), its main advantage is that it leads to more efficient planning algorithms due to the efficient representation of states as well as the avoidance of the frame problem occurring due to the frame axioms. PDDL describes the agent's actions as schemas, where a family of actions is represented by the common preconditions (that should hold in order for it to be executed) and effects that it brings about within the domain. As a consequence, PDDL might be considered more compact than the $\mathscr{A}$ language [Gelfond and Lifschitz, 1998] mentioned in Section 2.1.2.1.

Hierarchical task network (HTN) planning [Ghallab et al., 2004] is similar to classical AI planning but differing in that the objective is to perform a set of abstract tasks. In this, a set of atoms defines the state of the world and an action is the equivalent of a state transition. In addition to a set of operators similar to those of classical planning, however, the planning domain contains methods. These indicate how a task is broken down into smaller tasks called subtasks. Unlike in classical planning where problem specification defines a goal expression to be achieved, in HTN it indicates a non-primitive goal task to be achieved. The best established formalism for describing HTN domains is the one used in the SHOP2 planner [Nau et al., 2003]. The SHOP2 language is based on LISP syntax and its formalism comprises the elements below:

- Task - A task dictates an activity to execute. Tasks can be primitive (to be executed by a planning operator) or compound (comprising of other tasks).
- Operator - Operators, similar to PDDL actions, define how primitive tasks can be executed.
- Method - Methods indicate how to break down a compound task into sets of subtasks[3]. Methods have three parts: the task to which the method corresponds, the precondition (conditions that should be satisfied for the method to be applicable, formed by conjunctions, implications, quantifiers, etc. as well as external function calls) and the subtasks that should be achieved in order to pursue the task. Methods do not have effects.
- Axioms - Axioms, containing head and tail state facts that can be implicitly derived at each state.
- External function calls - External functions can be used, for example, to perform numeric evaluations.

[3] Planners like SHOP2 can manage partially ordered sets of tasks, having in this way plans interleaving subtasks from different tasks.

While planning research seem to be advancing in several directions (i.e. conditional planning, probabilistic planning, temporal planning, scheduling), in the last decade there has been a tendency to impose control rules in the planning process. Control rules [Bacchus and Kabanza, 2000] are formulas expressed in Linear Temporal Logic (LTL) which, applied to the (forward chaining) search, reduce the search space by pruning paths that do not comply with them. PDDL 3.0 [Gerevini and Long, 2006] follows the current tendency towards control rules over execution paths. PDDL 3.0, an extension of PDDL, imposes strong and soft constraints expressed in Linear Temporal Logic formulas on plan trajectories as well as strong and soft problem goals on a plan.

2.1.2.3 Planning in BDI Agents

Plans are a central element of a BDI agent implementation, as they indicate to the agent to proceed with the execution of specific actions in order to achieve the desired world state. In some architectures plans pre-exist in libraries and are instantiated through self-contained procedures (Jason [Bordini and Hübner, 2006], Jadex [Braubach et al., 2005], JACK [Winikoff, 2005], 2APL [Dastani, 2008], 2OPL [Dastani et al., 2009a]), in some (GOAL [Hindriks, 2009]) the agent follows action selection strategies, while in others a planner constructs a plan to be followed at execution time, allowing in this way more flexibility. In this section we discuss some recent approaches created to incorporate different types of planning mechanisms in BDI systems.

In [Meneguzzi and Luck, 2008] Meneguzzi and Luck try to tackle the problem that sometimes occurs in BDI systems, where a plan leads to failure, leading to the assumption that a goal is unachievable (which might be misleading since a different plan in the plan library might be able to reach it). The authors extend AgentSpeak(L) by adding a planning component that reasons about declarative goals, calling the new framework AgentSpeak(PL). The planning component is invoked (through a regular AgentSpeak action) when there are no appropriate plans in the plan library and forms high-level plans consisting of plans that already exist in the plan library. The way it does this is by evaluating the impact of existing procedural plans on the agent's beliefs (through their additions and deletions), transforming them into STRIPS operators and feeding them to a classical planner together with the agent's belief base and its goal state. If the planner produces a plan, this is transformed into an AgentSpeak plan and added to the plan library so that it gets executed. Additionally, in [Meneguzzi and Luck, 2009a] the same authors present a plan reuse strategy for AgentSpeak(PL) by providing an algorithm that generates context information (a set of conditions) for each plan, under which the plan will be able to be executed successfully. In this way plans might be reused under specific situations improving the performance of the agent. While an interesting approach, the main disadvantage of this is that it is based on the assumption that the plans in the agent plan base abide by some restrictions, that is, they should be expressed in terms of belief additions and deletions, in order

to be able to be transformed to STRIPS actions, making this a complicated task for the agent designers.

An innovative methodology to combine the BDI architecture with HTN planning is found in [Sardina et al., 2006]. The authors underline the similarities between the two, showing a mapping of the elements of the BDI and HTN systems (for example an action in BDI can be seen as a primitive task in HTN, or, a plan rule in BDI as a method in HTN). The proceed to present their framework, namely CANPPLAN, which is based on and extending the CAN (Conceptual Agent Notation) BDI agent language [Winikoff et al., 2002] and AgentSpeak [Rao, 1996]. The agent's configuration is specified by tuples and the system's transitions can be of two main types, namely *bdi* and *plan labelled transitions* and they are done through concrete derivation rules. A special construct called *Plan* provides the functionality of a built-in HTN planner. The authors conclude by showing how planning can be performed both for procedural as well as for declarative goals and discuss implementation issues.

The same authors, in a different line of work [De Silva et al., 2009], use HTN planning to compute "ideal" *abstract* (or, alternatively, *hybrid*) plans within BDI agents. They do so by transforming (and mapping) the BDI event-goals to *abstract operators* and then executing an HTN planner over these operators. Though the resulting plan might be valid, it is not necessarily the "best", since it might contain unnecessary actions that only complicate the execution without contributing to the achievement of the goals. Therefore, the authors come up with the notion of *minimal non-redundant maximally-abstract* plans to characterise the "ideal" hybrid plans and proceed to design an algorithm that computes such a plan, given a "non-ideal" hybrid plan computed by the planner, a hybrid planning problem and a decomposition tree.

All the above approaches present interesting features. The main idea behind them (whether using classical or HTN planning) is to take some of the common elements used in the respective BDI agent architecture (plan constructs or plan rules, belief base, etc.) and map them to a planning domain and problem. Then, feed this to a planner and use the result as a valid plan for the agent to follow. Such a methodology serves as an inspiration to our work, where in a similar fashion, the agent's deliberation methodology consists of translating the agent knowledge to a planning problem to be solved.

2.2 Social Structures and Agent Societies

The idea of building agent systems which capture some notion of what we as humans would call a society has been studied extensively, and has formed the basis for a large part of current research into the engineering of multi-agent systems. The notion of agent societies stems partly out of the desire to build systems which embody many of the perceived benefits of human societies and partly out of the wish to integrate

agent systems with the rapidly growing establishment of electronically-based human societies associated with the growth of the Internet and the social networks.

The role of a society is to permit its members to operate in a common environment and go after their respective objectives while coexisting and/or collaborating with others. Thus, social structures [Moses and Tennenholtz, 1995; Shoham and Tennenholtz, 1995] define a social level where the multi-agent system can be considered as a collection of entities that enhance the coordination of agent activities through structured patterns of behaviour [Vázquez-Salceda, 2004]. A social structure might contain roles, interaction patterns and/or communication protocols, a communication language and norms. Regulations and norms indicate the desirable behaviour of members. They are put in force by institutions that frequently have a legal standing and thus provide legitimacy and safety to members, decreasing at the same time the combinatorial explosion in agent interactions, as they put forward or even impose guidelines on the activities of the agents [Dignum, 2004].

When it comes to actually building agent societies, the environment and the constraints it places on an agent's execution might vary. Designers of agent systems must typically make assumptions about, or place constraints on the underlying environment in which their agents operate. In [Davidsson, 2000, 2001] Davidsson studies the types of constraints in the context of artificial societies and produces a broad taxonomy for existing approaches to the development of agent societies based on characteristics such as *openness*, *trust*, *stability* and *flexibility*. The reality of how open a given society will be depends largely on the choices of its designers and implementers. Davidson distinguishes four types of societies, according to the aforementioned characteristics:

1. *Open societies*: loosely structured societies with few or no restrictions over the action and interaction between the members (the author compares them to *anarchic societies*). They are normally characterised by openness and flexibility but also lack of trust.
2. *Closed societies*: societies restricting the access to external individuals, in which normally members have cooperatively pursue a goal. They are normally characterised by trustfulness but also lack of flexibility.
3. *Semi-open artificial societies*: societies that are controlled by an institution that acts as a gate-keeper, making assessment of entities that ask to join before it allows them to. Davidsson claims that they have greater potential for stability and trustfulness.
4. *Semi-closed agent societies*: societies to which access is limited for external individuals, but the members of which are able to create new members inside it. According to Davidsson they are less flexible than semi-open societies, nevertheless they show greater capacity for stability and trustfulness.

In [Jones et al., 2013] the authors distinguish and analyse three specific social concepts in socio-technical systems: *trust*, *role* and *normative power*. They suggest a synthetic

method towards engineering intelligent socio-technical systems. This consists roughly of three phases: *theory construction* (representation of a set S of observed social phenomena), *formal characterisation* (conceptual analysis expressed in a formal language or 'calculus', that is, a symbolic representation language of some kind) and *principled operationalisation* (tools employed in moving from a computational framework to a system platform implementation).

In principle, there are two main areas which approach social systems, *organisations* and *institutions*. While closely connected, the two have varying definitions amongst scientific literature.

North [North, 1990] makes a distinction between institutions and organisations. While institutions are abstract entities that define sets of constraints, organisations are instances of such abstract entities. The parties are members of an organisation (not members of an institution) that should follow the *institutional framework* defined inside the organisation.

A more specific distinction is given by Hogson in [Hodgson, 2006]. According to him, organisations are special institutions that involve:

1. Criteria to establish their boundaries and to distinguish their members from non members
2. Principles of sovereignty concerning who is in charge
3. Chains of command delineating responsibilities within the organisation

We analyse and discuss the different aspects of organisations and institutions in the following sections.

2.2.1 Organisational Models

As work interactions between people and enterprises are moving towards 'virtual' enterprises in which the different parties have autonomous activity, efficient communication and coordination between units and the design of equitable structures and processes are becoming essential in the well-being and the development of an organisation [Ancona et al., 2003]. In order to achieve sustainable adaptability and advantage to the environment, organisational models specifying the structure of societies have appeared, and during the last years they have played an important role in the design of information systems.

There are several definitions of what an organisation exactly means. Indeed, the word "organisation" is a complex word that has several meanings. In [Gasser, 1992], Gasser proposed the definition of organisation to which we subscribe:

"*An organisation provides a framework for activity and interaction through the definition of roles, behavioural expectations and authority relationships (e.g. control).*"

According to Ferber et al., several main features of organisations can be derived from the various definitions in the literature [Ferber et al., 2004]:

1. An organisation constitutes of agents (individual members) that exhibit some behaviour.
2. The overall organisation may be split into partitions that may overlap (also called partition groups).
3. Agent behaviours are functionally associated with the general organisation activity (notion of role).
4. Agents are engaged in dynamic relationships (also called *patterns of activities* [Gasser, 1992]) which may be "typed" using a classification of roles, tasks or protocols, describing in this way a form of supra-individuality.
5. Types of behaviours are connected via relationships between roles, tasks and protocols.

2.2.1.1 The Concept of Role

Role theory [Hindin, 2007] has been generally concerned with the way individuals of particular social positions manifest patterns of behaviour as well as the way other individuals are expected to behave within context-specific situations. In a social environment, individuals take *role positions* and their performance is determined by social norms, demands and rules, while social values will determine which norms and rules will be directed to what role positions.

Given the significance of the social aspect in organisational modelling, an important element when considering agent design is the concept of *role*. A role is a description of an abstract behaviour of agents. A role might describe the constraints (obligations, requirements, skills) that an agent will have to satisfy to obtain a role, the benefits (abilities, authorisation, profits) that an agent will receive in playing that role, and the responsibilities associated with it. It can also be the placeholder describing the templates of interactions in which an agent enacting that role will have to perform[4].

Much of the work in the relevant literature place roles in the core of the organisational description. While describing the Gaia framework in [Wooldridge et al., 2000], Wooldridge , Jennings and Kinny associate *responsibilities*, *permissions*, *activities*, and *protocols* to roles and propose a practical definition of computational organisations based solely on various interacting roles:

"*We view an organisation as a collection of roles, that stand in certain relationships to one another, and that take part in systematic institutionalised patterns of interactions with other roles*."

[4]In our thesis, we do not distinguish between role and role assignment as in [Odell et al., 2003]

In [Dignum, 2004] Dignum explains how a society's objectives can be broken down into role objectives. Then, the whole society can be considered as a super-role, the objectives of which are represented by organisational roles. The way these objectives are achieved depends on the requirements and characteristics of the specific domain. By specifying collaboration patterns, role models define high-level relationships between society members without fixing a priori the complete interaction process.

Dignum also explains how roles can be organised into *groups*, as a way of referring to a set of roles. She shows how useful this in an *interaction scene*[5] where a participant can be enacting one of several roles. She underlines the importance of dividing roles into groups as this allows to specify norms that must hold for all enactors in the group. This way, roles and groups of roles permit to 'split' the society objectives into role objectives, delegating in this way norms to enactors of roles in the group.

Finally, it is possible to define *role dependencies* between roles for the realisation of some objectives. This permits to indicate high-level dependencies between society members without necessarily specifying details of the type of coordination needed to distribute and achieve the objectives between them.

2.2.1.2 Organisational Frameworks

There are several organisational frameworks in the literature, but in this section we will review the most relevant to our work, as they include some sort of norms: MOISE [Hannoun et al., 2000] and OperA [Dignum, 2004].

MOISE (Model of Organization for multI-agent SystEms) [Hannoun et al., 2000] presents an organisational model for multi-agent systems, focusing on the aspect of roles (which constrain the action possibilities for each agent and define the activities that an agent can perform), organisational links (which regulate the interactions that the agents can have between them) and groups (which define who can cooperate with whom). Further to this, in [Hübner et al., 2002] the authors present the Moise$^+$ framework, an extension of MOISE. The main contribution of this extension is the distinction of the three aspects of an organisation used separately in MOISE: the *structural aspect* (including roles structure and inheritance, group clustering of roles and different types of links such as compatibility, authority, acquaintance and communication between agent roles), the *functional aspect* (including global plans, tasks etc.) and the *deontic aspect* (including norms, laws etc) of the model. While MOISE focuses on the organisational structure providing no particular normative layer, Moise$^+$ gives more emphasis on the complex relationships between actors as well as their rights and duties[6].

[5] An *interaction scene*, or *scene*, normally refers to a set of dialogical activities between agents in the same way theatre scenes are played. Agents (actors) engage in dialogs with respect to their assigned role (character).

[6] An extension of Moise$^+$ with a more powerful notion of norms, called Moise^{Inst}, is described in Section 2.2.2.3.3.

OperA, described by Dignum in [Dignum, 2004], is a framework for specifying organisations founded on the concept of social contracts. The OperA framework is composed of three components, Organisational Model (OM), Social Model (SM) and Interaction Model (IM):

- The OM describes the abstract specification of the organisation. It consists of a social structure (roles and the dependencies between the agents) and an interaction structure (describing possible interactions between agents). Additionally, a normative structure describes role norms (normative expressions that apply to roles) as well as scene and transition norms (expected conduct within a scene and restrictions over the transition from a scene to another, respectively).
- The SM describes the role enactment by the agents. This is done via social contracts, in a way that whenever an agent enacts a specific role, then the agent is adopting the terms of the contract associated with it. These terms represent the agent's responsibilities within the framework.
- The IM specifies the interactions between the agents. This is done via pre-specified scenes (not defining how objectives can be achieved as this is left to the designer's choice) and *landmarks* (partially ordered descriptions of desirable intermediate states) to be brought about within the scenes.

One of the strengths of OperA (being a model applicable to both human and software agents) is also the main limitation, as the framework provides no support to (semi-)automatically generate OperA-aware software agents (two frameworks based on OperA, namely OMNI and ALIVE, partly provide such functionality and will be discussed in Section 2.2.2.3.3).

For a long time the notion of organisations and the notion of institutions were two distinct, separate views on modelling MAS governance. Nevertheless, in [Ferber et al., 2004] Ferber et al. proclaim that an organisation is made of two aspects: a *structural aspect* (also called static aspect) and a *dynamic aspect*. The structural aspect of an organisation consists of two parts: a *partitioning structure* and a *role structure*. A partitioning structure shows how agents form groups and how groups are related. A role structure is defined, for every group, by a set of roles and the relations between them. This structure defines also the set of restrictions that agents should comply with to play a particular role and the benefits associated with that role. The dynamic aspect of an organisation is associated with the institutionalised templates of interactions that are specified within roles (norms). According to Ferber et al. it additionally defines the modalities to *create*, *kill*, *enter groups* and *play roles*, the way these are applied and the way organisation subdivision and role structuring are associated with an agent's behaviour.

The last points form part of the *institutional* aspect of the organisation. While organisations provide the main concepts for an abstraction of social structures, they lack coordination frameworks that imitate the coordination structures within an organisation. That is where institutions come into force. Institutions enforce the organisational

aims of the agent society and dictate a performative structure (i.e. a description of how scenes are interconnected through different types of transitions and how agents, via different roles, participate in these scenes) and a dialogical framework between members of the society [Noriega and Sierra, 2002; Sierra et al., 2001]. The benefit of institutions lies in their ability to provide legitimacy and security by providing social conventions and setting up a normative framework for their members. We explain more about institutions in Section 2.2.2.

2.2.2 Institutions

As explained in the beginning of Section 2.2, in order to control the real-world environment, humans have created moral and/or legal structures that deal with specific aspects of daily lives, giving some sense of order. Human law therefore provides rules and guidelines which are enforced through social institutions to regulate behaviour.

The problem that occurs when automated systems are put into function under real-world conditions is that human regulations are usually expressed in a quite abstract fashion and often allow various interpretations[7] [Grossi and Dignum, 2005]. The fundamental incentive for this is that they are created to cover a large number of cases with the same legal content and to maintain regulations solid and valid over time. In law, it might even be up to the judges to interpret the laws with respect to some specific context and decide whether they were violated or not. Although it might be facilitating for humans, the abstraction and possibility of more than one interpretation poses serious problems when trying to practically apply such regulations in computational systems, where meaning should be accurate and unambiguous.

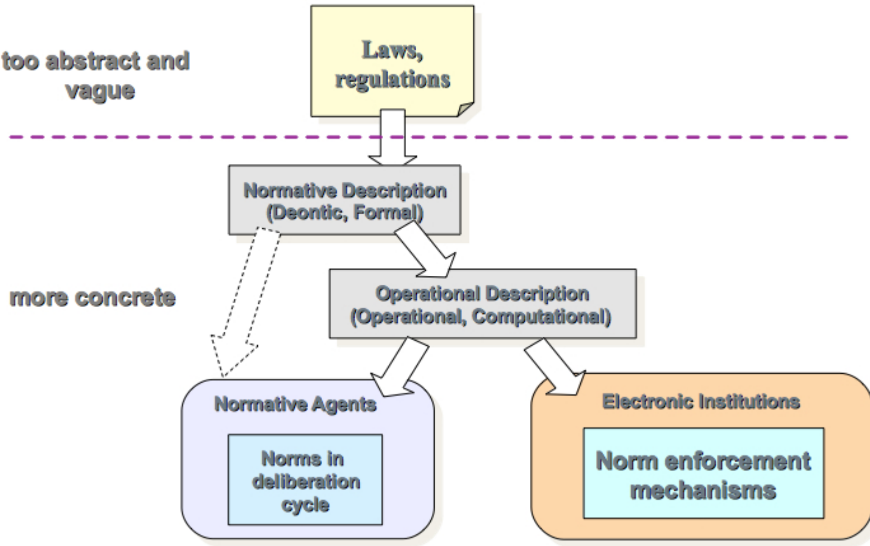


FIGURE 2.2: From human laws to their electronic representation

[7]In Section 2.2.2.2.3 a review of different norm formalisations is provided.

On the other hand, with the agent paradigm having emerged in the last decades, several social interaction systems, where agents represent individual's interests, have seen the light. Since agents might not have similar interests, objectives and conflicts between them might occur and therefore, the necessity for regulations to be present and enforced in such systems is strong. In order for this to happen, it is essential to have a shared understanding of normative concepts between interacting agents and the environment in which they operate. Additionally, the abstraction of the high-level law descriptions should be made concrete through an operational level that provides machine-readable specifications to be unambiguously interpreted and applied.

For norms to be represented in computational systems, high-level specifications are translated down to a computational level, possibly including formal ontological descriptions and given formal deontic semantics. Such translations can be strongly context dependent. Taking it a step further, reductions from the deontic layer to operational descriptions act as a bridge towards direct implementations, treatable by computational systems [Vázquez-Salceda, 2004; Aldewereld, 2007]. Seen from the institutional perspective, translating norms from abstract deontic facts to machine readable rules allows for norm enforcement mechanisms to be developed and applied. On an individual level, creating operational rules from abstract deontic specifications, enables them to be taken into consideration in its decision making process, regulating its behaviour in the environment in which it operates. Figure 2.2 visualises how abstract norms are broken down to various levels of normative descriptions and operational rules when it comes to being represented, analysed and reasoned about in automated systems. An important aspect of Figure 2.2 is that to avoid semantic mismatch between individual agent interpretation of the norms' specifications and their institutional interpretation, the operational description used should be concrete and provide a uniform understanding of the normative concepts.

2.2.2.1 Human vs. Electronic Institutions

Institutions are identified by the set of social constraints that govern the behaviour and relationships between members of a society. This concept of *human institution* can be used and applied to agent-based interactions, in *Electronic Institutions*. An Electronic Institution is a multi-agent system where agents' behaviour is governed by a set of published norms, rules or regulations which bring about a set of expected behaviours for agents interacting in a social context.

Nobel laureate Douglass North [North, 1990] studied the effect of sets of constraints, that he refers to as *institutions*, on the behaviour of human organisations, focusing on their performance. North claims that institutional constraints facilitate human interaction, forming choices and allowing effects to be foreseen. The creation of *institutional constraints* allows for a growth of the complexity of the organisations while keeping reduced interaction costs, by ensuring trust between parties and giving shape to their

choices. Equally important, it allows the participants of the institution to act, and expect others to act, according to a list of rights, duties, and protocols of interaction.

Therefore, the creation of institutions provides trust among parties even when they do not have much information about each other[8]. In environments with incomplete information, cooperative interactions can perform ineffectively unless there are institutions which provide sufficient information for all the individuals to create trust and to control unexpected deviations on interaction.

Institutions can be classified according to how they are created and maintained, or on the formality of their rules. In the former case, institutions can be created from scratch and remain *static* or be continuously *evolving*. In the latter, institutions can be *informal*, that is, defined by informal constraints such as social conventions and codes of behaviour, or *formal*, defined by *formal rules*. Formal rules can be *political and judicial rules*, *economic laws*, or *contracts*.

In *formal institutions* the purpose of formal rules is to promote certain kinds of exchange while raising the cost of undesired kinds of exchange. Elinor Ostrom [Ostrom, 1986] classifies formal rules into 6 types:

- *Position rules*: to define a set of positions (roles) and the number of participants allowed for each position.
- *Boundary rules*: to state how participating entities are picked to hold or leave a position,
- *Scope rules*: to determine the outcomes and the external values (costs, inducements) related to them.
- *Authority rules*: to define the set of actions associated with a position at a certain node.
- *Aggregation rules*: to specify decision functions for each node to map actions into intermediate or final outcomes.
- *Information rules*: to establish channels of communication between participating entities and state the language in which the communication will occur (the protocol).

As norms are in essence what characterises institutions, they do not only serve as norms to be complied with, but also serve as indications to predict other norms that could be applicable.

As explained earlier in this section, in environments with incomplete information, institutions hold a substantial role in order to establish trust between members. This statement holds both for closed multi-agent systems, where trust is implicit, and open multi-agent systems, where trust is something that has to be built. For the latter type of systems, *Electronic Institutions* (*e-institutions*), seen as computational realisations of

[8] No institutions are necessary in an environment where parties have complete information about others, as trust and reputation systems are enough to bring some order in their interactions.

traditional institutions, can be considered as a comprehensible modelling approach [Esteva, 2003; Sierra et al., 2004]. Electronic Institutions can be seen as a regulated virtual environments in which the respective interactions between participating entities occur.

More formally, according to Vázquez-Salceda [Vázquez-Salceda, 2004] "*[...] an Electronic Institution is the model of a human institution through the specification of its norms in some suitable formalism(s). The essence of an institution, through its norms and protocols* can *be captured in a precise machine processable form and this key idea forms the core of the nascent topic of institutional modelling.*"

Vázquez-Salceda [Vázquez-Salceda, 2004] states that institutions, seen as norm providers and enforcers, are intended to solve the following issues in the context of multi-agent systems, making in this way agents more successful in the accomplishing of their objectives, since they:

- Lessen uncertainty about other agents' conduct within the institution.
- Decrease misunderstanding with a shared set of norms ruling interactions.
- Allow agents to predict the result of some specific interaction between participants.
- Make the every agent's decision-making simpler by reducing the number of actions that can be performed.

While the above is useful when analysing and modelling social systems, in this thesis we do not put substantial emphasis on the differentiation between organisations and institutions. As our objective is to work with the operational and functional aspects of multi-agent systems, we choose to focus on the institutional perspective and to view organisations as a purposeful structure within a social context. A list of institutional frameworks can be found in Section 2.2.2.3.

2.2.2.2 Norms

In settings where multi-agent systems are used to implement a system with a specific goal one does not want a member's behaviour to diverge from the overall goal of the system. In order to bound the autonomy of the agents in such situations and guarantee a certain behaviour of the overall system one has to construct agent organisations by allowing interactions between the agents in these organisations. The institutional interpretation of reality and the rules of behaviour for the agents within the organisation are described using *norms* and most of the descriptions of institutions are based on the formal definition of their norms and their effect in the actions performed by the agent members. In this section we discuss the different approaches when modelling norms as well as the issues involved in the process.

Regulations have been studied from many different points of view, including Sociology and Law. Specifically, within the Artificial Intelligence community there has been

a continuous effort on researching how to create efficient formalisations of regulations and norms from a logic perspective. As a consequence, a lot of work on legal and *normative systems* formalisation (see Sections 2.2.2.3 and 2.2.2.4) has been produced. Such work is declarative in nature and gives emphasis on the expressiveness of the norms, the establishment of formal semantics and the verification of consistency of a given norm set.

A *norm* can be defined from several perspectives [Vázquez-Salceda, 2004]. According to Vázquez-Salceda, it can be seen as a rule or standard of behaviour shared by members of a social group; as an authoritative order or standard by which something is judged and, on that basis, approved or disapproved; as standards of right and wrong, beauty and ugliness, and truth and falsehood; or even as a model of what should exist or be followed, or an average of what currently does exist in some context. In [Vázquez-Salceda, 2004], the same author makes a distinction between the treatment of norms in different contexts: *Social Theory*, *Legal Theory*, and *Computer Science*.

Scott [Scott, 1995] defines a framework to describe the interactions between institutions and organisations. In this framework, an institution is composed of the regulative aspects, the normative ones, and the cultural-cognitive ones. In Scott's view, there is a distinction to be made between norms, which are related to moral aspects and describe *regulative aspects* of individual behaviour, and rules, which are related to coercive aspects and describe *prescriptive, evaluative, and obligatory aspects* of social life.

Moreover, for Scott, roles are conceptions of appropriate goals and activities for particular individuals or specified social positions, creating, rather than anticipations or predictions, prescriptions, in the form of normative expectations, of how the actors fulfilling a role are supposed to behave. These roles are directly related to those norms that are not applicable to all members of a society, but to selected types of actors.

2.2.2.2.1 Regulative vs. Constitutive Norms

The main question that rises from the above is how norms might be represented in regulated computational systems. According to many studies in legal and social theory, normative systems consist of regulative as well as non-regulative components [Searle, 1969; Jones and Sergot, 1996; Searle, 1997].

Regulative rules specify commitments, restrictions and authorisation over states of affairs (e.g. "a client must pay when a purchase is made") in social systems and are mainly taken into consideration in practical reasoning. A common way of modelling regulative norms is through various types of *Deontic Logic*, that is, formal systems devoted to capturing the concepts of *obligations*, *prohibitions* and *permissions*. In Section 2.2.2.2.3 we will provide a detailed analysis of the most common deontic formalisations.

Constitutive rules on the other hand, concern *what counts as what* in a given institution (e.g. "visa payment counts as payment"). The paradigmatic syntax of constitutive rules has been taken to be, since [Searle, 1969, 1997], the form of "counts-as" statements:

[...] "institutions" are systems of constitutive rules. Every institutional fact is underlain by a (system of) rule(s) of the form "X counts as Y in context C" [Searle, 1969].

In [Searle, 1969, 1997] Searle opines that no institution is given without constitutive rules and therefore without "counts-as" and that such statements are stated relatively to a context because *what counts as what* differs from institution to institution.

A significant contribution towards the study of formal aspects of the least explored kind of norms, that is the constitutive norms, is [Jones and Sergot, 1996], which is the first study of the way "counts-as" conditionals can be modelled[9]. In addition to this, in [Grossi, 2007] Grossi focuses on a formal analysis of institutions and especially constitutive norms, specifically the counts-as relation between abstract and concrete terms. Grossi presents a formal language based on description logic for the expression of institutions as an imposition of institutional terminology upon brute terminology. This imposition is realised through the specification of counts-as definitions that relate normative concepts to concepts in the (institutional) reality, linking in this way abstract norms to concrete situations. In concrete, institutions state terminologies (i.e. different and possibly inconsistent sets of contextual concepts) and counts-as statements are statements talking about these terminologies. Thus, by means of a counts-as operator, it is given that a certain concrete concept can be considered an instantiation of an abstract concept used in the norms. Grossi provides no implementation of his formal analysis, however, he makes a comparison between the use of regimentation and the use of enforcement of the norms in e-institutions.

Other more practical works on counts-as rules include Álvarez-Napagao's PhD thesis, which provides a practical implementation of a reasoner that uses counts-as rules, and the results can be found at [Álvarez-Napagao et al., 2010; Aldewereld et al., 2010]. In Section 2.2.2.4.1 some agent frameworks including constitutive norms are described.

2.2.2.2.2 Institutional and Normative Power

There exists a lot of literature on legal concepts such as power, right, duty, commitment, enforcement etc., deriving from Legal Systems. In this section we discuss some of these concepts and especially *institutional power*, which captures the notion of how agents might assume the "power" to bring about affairs within an organisation, and different attempts to give semantic meaning and formally capture it.

Hohfeld is the first to make a distinction between the *legal power* (the power to exercise a legal privilege), the *physical power* (physical ability to perform) and the *privilege* (as the opposite of *duty*) to do something [Hohfeld, 1917]. Later, based on Hohfeld's

[9] More on this work will be explained in Section 2.2.2.2.2.

consideration, Jones et al. in [Jones and Sergot, 1996; Jones et al., 2013] claim that an agent might have the practical possibility without being empowered. Therefore, the practical ability to perform acts which lead to a normative state, is neither a necessary or sufficient condition for being empowered. Jones et al. proceed to introduce the notion of *institutionalised* (or *institutional) power* to capture the 'legal power' or 'empowerment' for particular entities to perform specified types of acts, within specific contexts. They reason about why institutionalised power cannot be considered equivalent to permission or practical possibility to perform some action.

Motivated by their desire to 1) formalise the idea that within a specific institution, certain acts *count as* means of bringing about kinds of normative states of affairs, and 2) capture the fact that there are usually restrictions within an institution which state that some states of affairs of a given type *count as* or are to be *classified* as states of affairs of another given type, they proceed to conceptualise institutional power in [Jones and Sergot, 1996]. With $\mathrm{E}_x A$ standing for "x sees to it that/brings it about that A" and with the "counts as" reading for the conditional $\Rightarrow_s$, they define statements of the type $\mathrm{E}_x A \Rightarrow_s \mathrm{E}_x F$. Such a statement then reads as: "relative to institution s, x's act of seeing to it that A counts as, or is to be regarded as, an act of establishing the state of affairs F, performed by x". Based on this, they then build with inference rules and axioms a modal language for representing the notion of empowerment.

Oren et al. [Oren et al., 2010] on the other hand, present a model of 'normative power'. This may refer to the ability to *create*, *delete*, *change* a norm. They model normative power as a tuple: $\langle Mandators, Context, Pre, Post \rangle$ where *Mandators* are agents in the system, *Pre* and *Post* contain norms to be removed and inserted into the system by the application of the power. They then proceed to explain how the application of the power on the norms within affects the institutional environment within which they exist and also define powers over powers (delegation of a power to someone else).

2.2.2.2.3 Languages to Express Regulative Norms

In legal theory, norms are always expressed in natural language. That makes them ambiguous and hard to deal with in computational systems. To solve this gap, Mally intended to create an exact system of pure ethics to formalise legal and normative reasoning [Mally, 1926; Lokhorst, 1999]. That was the first attempt at creating a *Deontic Logic*, based on the *classical propositional calculus*. Von Wright [von Wright, 1951] presented a formalism based on *propositional calculus* that was similar to a normal modal logic. Although traditionally used to analyse normative reasoning in law, Deontic Logic (alternatively known as the logic of normative concepts) is also applied beyond the domain of legal knowledge representation and legal expert systems; other applications include *authorisation mechanisms*, *electronic contracting* and more. Deontic Logic has been invented to explore the properties and dynamics of norms over time and uses special modal operators to represent logical relations among *obligations*, *permissions* and *prohibitions*. Adapting Von Wright's proposal, the *Standard System of Deontic*

Logic (SDL) or *KD* was created as a modal logic with the following axioms [von Wright, 1951]:

$$
\begin{array}{ll}
O(p \rightarrow q) \rightarrow (O(p) \rightarrow O(q)) & \text{(KD1 or K-axiom)} \\
O(p) \rightarrow P(p) & \text{(KD2 or D-axiom)} \\
P(p) \equiv \neg O(\neg p) & \text{(KD3)} \\
F(p) \equiv \neg P(p) & \text{(KD4)} \\
p, p \rightarrow q \vdash q & \text{(KD5 or Modus Ponens)} \\
p \vdash O(p) & \text{(KD6 or O-necessitation)}
\end{array}
$$

where O, P and F are modal operators for *obliged*, *permitted* and *prohibited*.

Thus Deontic Logic allows for expressing norms as obligations, permissions and prohibitions. Standard Deontic Logic is expressive enough to analyse how obligations follow each other and is useful to find possible paradoxes in the reasoning. Verbs such as *should* or *ought* can be expressed as modalities: *"it should happen p"*, *"it ought to be p"*. The semantics of the O, P and F operators define, for a normative system and in terms of possible worlds, which situations can be considered as ideal, i.e. situations where no norm is neglected or unfulfillable. However the KD system lacks operational semantics, making it not possible to be directly used in computational and/or reasoning systems to decide over the course of action.

Working on the formalisation of agency, Kanger [Kanger, 1972] introduced an operator of the form $Do(a, p)$ where a represents the agency and p a propositional expression. He adopted the notion of "*sees to it that (stit)*" to express that a sees to it that p happens. Other operators have also been added to deontic logic by several works. In the Kanger-Lindahl-Pörn logical theory [Kanger, 1972; Kanger and Stenlund, 1974; Pörn, 1974] the operator E indicates the expression of direct and successful operations. $E_i A$ means that an agent i *brings it about* that A (i.e. agent i is directly involved and makes A happen). Santos, Jones and Carmo later brought the operators G and H to formalise indirect actions [Santos et al., 1997]. $G_i A$ says that an agent i *ensures* that (but is not necessarily involved into achieving) A. Respectively, $H_i A$ says that an agent i *attempts to make it the case* that A.

Von Wright proposed an extension of KD Deontic Logic called *Dyadic Deontic Logic* [von Wright, 1951, 1956]. In this, *conditional obligations* can be expressed as $O(p|q)$ (meaning that it is obligatory that p if q) and *conditional prohibitions* as $P(p|q)$ (meaning that it is permitted that p if q). Some other researchers combine deontic logic with temporal aspects, resulting in *Temporal Deontic Logic* [Brunel et al., 2006]. In the literature it is often written as $Obg_i(a < p)$ to indicate that "It is obligatory for i to perform α before p holds" [Demolombe and Louis, 2006]. Brunel et al. also specify that it is obligatory to satisfy ϕ before k time units as $O_k(\phi)$.

From a different perspective, and much more recently, Conte and Castelfranchi try to use deontic restrictions in order to guide the agent's behaviour. In [Conte and Castelfranchi, 2001] they introduce a new operator *OUGHT* to the formalism's modal operators *BEL*, *GOAL*, *HAPPENS* and *DONE*, allowing to express normative concepts.

Unfortunately, none of the aforementioned deontic representing formalisms is entirely free of representational problems [Hansen et al., 2007]. Many paradoxes such as logical expressions that are valid in the logical system for deontic reasoning, but which are counter-intuitive from the human thinking perspective still exist. In order to deal with these troublesome cases of formalisation, one may focus on the pragmatic standpoint and use logics that are sufficient for specific situations. Even though such a logic might not always be adequate in general, one does not have to resolve all the deep problems that philosophy gives rise to for the general abstract cases.

2.2.2.2.4 Operational Semantics for Regulated Systems

While it is true that most of the studies in the literature define semantics to interpret norms, there are a few dealing with the connection between such semantics and the operational level closer to the actual practical implementation. Assigning operational semantics over regulative norms is not straightforward. Deontic statements express the existence of norms, rather than the consequences of following (or not following) them [Walter, 1996]. In order to implement agents and institutional frameworks capable of reasoning about norms, we need to complement deontic logic with semantics defining fulfilment and violation - among other operational normative concepts[10].

Consequently, when dealing with normative systems, where the expected behaviour of agents is described by means of an explicit specification of norms, some of the relevant issues that should be taken into consideration are the following (see Figure 2.2):

- There is a need to formally connect the deontic aspects of norms with their operationalisation, preserving the former.
- Ideally, the operational semantics (and descriptions) should be created in a way that ensures flexibility in their translation to actual implementations while ensuring unambiguous interpretations of the norms. For instance, the semantics used by a society compliance mechanism, and the semantics integrated in the reasoning cycle of its individual agents, must be aligned to avoid, e.g. the agent norm reasoning mechanism stating that no norm has been violated while the compliance mechanism states that the agent has violated some norm instance.
- There should be a way to represent permissive norms and their relation to obligations, and how norms evolve throughout time.

[10] In Chapter 3 of this thesis we present general operational semantics for norms including these aspects, which are then relived in Chapters 4 and 5.

- Possible penalties and consequences for norm breaking should be explicitly defined, making sure those who are bound to them are aware of the cost of such breaches.
- The operational description should be useful for both the agents that operate in the institutional context and for the institution itself. Ideally, from the agent perspective this would mean that the norms' operational descriptions can be directly parsed by the agents and automatically taken into account in the agent's deliberation cycle; from the institutional perspective this would mean that the same norms' operational descriptions can be translated into the computational mechanisms that will enforce norms in the environment.
- From a practical point of view, abstract norms have to be distinguished from their actual instances. For each abstract norm, many instances may happen during the norm's lifetime.

Some works in the literature (see Sections 2.2.2.3 and 2.2.2.4) present solutions that tackle the above issues separately, but, as we will see in the next sections, it is hard to find a proposal that manages to fill the gap left by all the practical issues at the same time.

Of course it is important that we also define how the agents use these norms to govern their behaviour as this determines the interaction between the individual agents and the multi-agent system.

However, there is no consensus of what norms are, how they should be modelled, and how agents should reason about them. Some issues still open in this direction are:

- How agents communicate, understand, and fulfil the other's expectations.
- How the collective behaviour emerges from the composition of individual behaviours.
- How to formalise normative systems in a computational representation.
- How to define mechanisms, similar to the ones existing in human societies, to ensure trust in open systems and complex environments.
- How the norms can be incorporated in an organisation, in a way that the individuals behave according to them.

2.2.2.3 Institutional View: Normative Multi-Agent Systems

There exists a rich research background in understanding the way legal (or normative) systems are put into function within human societies and what their influence on the activities of social members is. Norm-aware systems can be realised through *normative multi-agent systems* that merge both social norms and multi-agent systems.

Normative multi-agent systems offer the ability to integrate social and individual factors to provide increased levels of fidelity with respect to modelling social phenomena

such as cooperation, coordination, group decision making, organisation, and so on, in human and artificial agent systems [Boella et al., 2006].

We distinguish two categories of norm-based frameworks: 1) the ones that focus on the normative context (the institutional view), and 2) the ones that focus on the agent perspective and how the norms affect the decision making (the agent view). This section is an overview of some of the most important frameworks that cover the institutional aspect of normative multi-agent systems while Section 2.2.2.4 covers the normative agents from the agent's perspective.

2.2.2.3.1 Basic Concepts in the Modelling of Normative Multi-Agent Systems

Normative multi-agent systems consist of autonomous agents who must comply with social norms [Boella et al., 2008b; Dignum, 1999]. The meaning of complying with or fulfilling a norm might depend on the interpretation of analysers of a system. In societies with few members it is easy to see a norm as fulfilled if all agents have fulfilled it; however, in societies with more members, it can be sufficient for a percentage of the members to comply with a norm to see it as fulfilled.

While systems that are regulated by norms differ from one another, some generic attributes can be noticed [López y López et al., 2006]. They are composed of sets of agents that share sets of obligations, social laws or codes to obey. According to Lopez et al., in such systems is not realistic to expect that all norms are known in advance, since conflicts between agents may arise and, therefore, new norms might occur, and also, norm compliance cannot be expected, since agents might prefer to disobey or unwillingly fail to obey. Therefore, such systems need mechanisms to handle both the modification of norms as well as possible unexpected behaviour of autonomous agents. In [López y López et al., 2006] the authors conclude that normative multi-agent systems have the following characteristics:

- *Membership*. An agent must recognise itself as part of the society. By recognising this membership, agents can demonstrate their readiness to adopt and comply with social norms.
- *Social Pressure*. Authoritative power is meaningful only if norm violations are followed by sanctions application. Such a control could be coming from assigned agents and should be socially recognised.
- *Dynamism*. Normative systems must be of dynamic nature, allowing new norms to be put into force or abolished and members to join or leave the system. Such dynamism may result in unexpected behaviours or impact other members' behaviour.

Given these characteristics, [López y López et al., 2006] argues that multi-agent systems must include mechanisms to defend norms, to allow their modification, and to enforce them.

Andrighetto and Conte [Andrighetto and Conte, 2012] distinguish several types and subtypes of social norm adoption decision processes:

1. *Apparent adoption*, when the agent's goal is the same as what is being enforced by the norm.
2. *Instrumental adoption*, guided by rules that are thought to achieve the agent's goals, and which are broken into: diligent adoption (referring to technical norms), artificial adoption (being established by external goals), cooperative adoption (in order to pursue common goals), adoption by trust (established by the trust that one has towards other members and with the intent to be adopted by all the society's members), adoption by commitment (followed when something has been promised), conditioned adoption (caused by reciprocity between agents).
3. *Terminal adoption*, happening because it is in the agent's belief that norms have to be unquestionably obeyed.

The notion of *enforcement* is widely used when dealing with norms. Villatoro et al. [Villatoro et al., 2011] distinguish between two enforcing mechanisms, *punishment* and *sanction*. Punishment refers to the enforcement mechanism discouraging behaviours by modifying the costs and benefits of a particular situation, while sanction refers to the enforcement mechanism which notifies the violators that their behaviour is not acceptable and that they have caused infractions. The authors proceed to show how sanctioning is a more effective enforcing strategy than when the agents are being guided by the pure motivation to avoid punishment, and that when it comes to achieving cooperations, these become more stable and less costly.

2.2.2.3.2 Institutional Models for Normative Multi-Agent Systems

An example of a framework based on norm-aware agents is described by López y López in [López y López, 2003; López y López et al., 2001; López y López and Luck, 2002]. This framework defines normative agents as those agents whose behaviour is shaped by the obligations they must comply with and prohibitions that limit the kind of goals that they can pursue. To this extent the author defines a norm frame which includes *addressees* (the agents to whom the norm applies), *beneficiaries* (those who the addressees are focused on), *normative goal* (the goal that is to be achieved or avoided as specified by the norm), *contexts* (the states of the environment when the norm is active), *exceptions* (states when the norm is not active) and *rewards* and *punishments* (responses to the compliance or violation of a norm). This norm frame (describing what is supposed to be achieved/avoided in which context) is then linked to actions by means of a relation between a specific action and a norm. Either an action benefits a norm (making it possible to be compliant with the norm when the action is executed) or it hinders a norm (executing the norm makes it impossible to be norm-compliant). It is then stated that, for a normative agent, all actions benefiting the norms are permitted and all actions that hinder the norms are prohibited. In

[López y López and Luck, 2002] the authors define different stages through which a norm goes since it is established until it becomes abolished. This, seen as the norm's lifecycle, includes stages such as *issue*, *spread*, *adoption*, *modification*, *abolition*, *activation*, *compliance*, *reward*, *violation*, *sanction*, *dismissal* and *non-sanction*.

A framework for formally specifying electronic institutions, described as a *dialogic framework* by the creators, is AMELI, based on the ISLANDER formalism [Esteva and Sierra, 2002; Esteva et al., 2002]. This abstraction considers an agent-based institution as a *dialogic system* within which, the interactions are composed by different *dialogic activities*. The latter, also named *illocutions* [Noriega, 1997], are made up by agent group meetings, called *scenes*, that comply to well-defined protocols. Breaking up the interactions into scenes permits the framework to be characterised by modularity, similarly to other popular programming techniques such as Object Oriented Programming. Special *normative rules* capture actions' consequences through *illocution schemes* (illocution formulas containing possibly unbound variables and thus representing a set of possible illocutions) and determine when norms (obligations) are activated and get fulfilled. A feature of ISLANDER/AMELI is that they use regimentation in order to make sure that the norms are always followed. In other words, no deviation from or ignoring of norms by the agents is allowed. Another limitation is that the agent is not aware of the norms and there is an external mediator that filters unwanted behaviour. AMELI was the first fully implemented institutional framework providing tools able to parse ISLANDER specifications and execute them at runtime.

The HARMON*IA* methodology of [Vázquez-Salceda, 2004; Vázquez-Salceda and Dignum, 2003] proposes a different approach, by explicitly specifying the norms of the institution and keeping track of the refinement steps taken to track all the translations needed to implement the abstract norms of the institution. The framework distinguishes four different levels: an *abstract level* (which contains the abstract norms), a *concrete level* (containing concrete instantiations of the abstract norms), a *rule level* (where concrete norms are translated into rules that can be computed by the agents), and a *procedure level* (where the final mechanisms to follow the rules are implemented). The use of such a layered approach shows its benefits in situations when changes to the norms are required (which happens often in real-world institutions). Since connections between the implemented mechanisms and the abstract norms that they relate to are made explicit, a change to the (abstract) norms can be propagated through the system. HARMON*IA* also tries to solve (part of) the restrictive regimented nature of the ISLANDER formalism, by the proposal of Police Agents, agents which are responsible for the enforcement of the norms. This allows to control the system's safety (avoid non-desirable situations due to some agent's failure), while still providing the agents (enough) autonomy to perform their tasks in manners that were not thought of at design, therefore enabling them to handle unforeseen situations and adding a level of robustness to the system. There exists no implementation supporting this framework, but parts of HARMON*IA* have been included in OMNI (see Section 2.2.2.3.3).

In [Aldewereld, 2007], Aldewereld further extends the ideas of Vázquez-Salceda in [Vázquez-Salceda, 2004] by applying parts of the methodology to highly-regulated environments (environments governed by many of complex norms). The author makes a distinction between "substantive" (expressing wanted and unwanted situations and allowing the possibility of violations) and "regimented" (expressed as direct constraints on the agent's actions and therefore always ensuring compliance) norms and deals mainly with the first type. [Aldewereld, 2007] identifies four important aspects of institutional implementations: 1) an ontology to allow communications between agents, and to express the meaning of the concepts used in the norms; 2) an (explicit) normative description of the domain, specifying the allowed interactions in the institution, presented in a format readable by (norm-aware) agents; 3) a set of protocols (conventions) that agents that are incapable of normative reasoning can use to perform their assigned task; and 4) an active norm enforcement to see to it that the norms specified for the domain are adhered to and order and safety is guaranteed in the system. These four elements are combined into a framework that gives the relations between laws and electronic institutions. Moreover, (formal) methods are specified for the implementation of norm enforcement and the (automatic) creation of protocols (based on constraints specified by the norms). However, there is no implementation of this framework, although some concepts are included in OMNI and ALIVE (see Section 2.2.2.3.3).

In [Kollingbaum et al., 2008] the authors present a framework called Requirement-driven Contracting (RdC), for the automatic formation of executable norms from requirements and correlated relevant information. Requirements include *domain assumptions*, *preferences*, and *priorities*. With the help of templates and formal language constructs, the designer writes the environment specification (ES) to translate requirements into system objectives, domain assumptions into domain restrictions, and specifies preferences over objectives and priorities over clashing preferences. The last ones are put into the RdC algorithm to acquire an executable specification of norms that regulates, as a consequence, the virtual organisation.

2.2.2.3.3 Hybrid Organisational-Institutional Models for Normative Multi-Agent Systems

There is a recent trend in combining both organisational and institutional aspects in the same framework, in works such as Moise^{Inst} [Gâteau et al., 2005], OMNI [Dignum et al., 2004; Vázquez-Salceda et al., 2005] and the language NLP [Hübner et al., 2011].

Moise^{Inst} [Gâteau et al., 2005], seen by the creators as an institution organisation specification, is founded on the Moise^{+} [Hübner et al., 2002] organisational model and focuses on specifying agent rights and describing the duties of each society role through four types of specifications, *Structural* (SS), *Functional* (FS), *Contextual* (CS) and *Normative* (NS). Normative specification (NS) extends the Moise^{+} deontic specification and defines rights and duties of roles and groups on a mission (set of goals) and in

a specific context; Structural specification defines roles that agents enact and relations between these roles as well as an additional level (group) to which roles might belong and in which interactions take place; Functional specification defines all goals that have to be achieved; Contextual specification describes a set of contexts influencing the dynamics of the organisation as well as the transitions between contexts. The creators show how Moise^{Inst} may be used in interactive games, both on the agent operating layer (where avatars operate as autonomous agents) as well as on the multimedia game management and control layer (where an institutional middleware is dedicated to the arbitration and supervision of the whole organisation). While a powerful and expressive organisational framework, norm specification in Moise^{Inst} lacks conditional norms (only allows time limitations and special conditions in the norm definition). We are also not aware of any functional implementation of Moise^{Inst}.

Organizational Model for Normative Institutions (OMNI) [Dignum et al., 2004; Vázquez-Salceda et al., 2005] brings together some aspects from two existing frameworks: OperA [Dignum, 2004] and HARMON*IA* [Vázquez-Salceda, 2004; Vázquez-Salceda and Dignum, 2003]. OMNI is spread throughout three dimensions that describe different characterisations of the environment. The *Normative Dimension* of the organisation (specifying the mechanisms of social order, in terms of common norms and rules, that members are expected to adhere to), the *Organisational Dimension* (which describes the structure of an organisation, an can therefore be viewed as a means to manage complex dynamics in societies) and the *Ontological Dimension* (defining environment and contextual relations and communication aspects in organisations). In OMNI, the environment is represented in three levels: 1) the *Abstract* level which defines a high level system abstraction similar to the requirements analysis. It contains an ontology of the model describing all the different organisational terms such as norms, rules, roles, sanctions, etc.; 2) the *Concrete* level, supporting the design of the normative institution. Its normative dimension defines the norms and rules of the system, the organisational dimension defines the organisational structure and its ontological dimension defines concrete ontological concepts.; and 3) the *Implementation level*, where the design of the normative and organisational dimensions is implemented. In particular, mechanisms for role enactment, norm enforcement, a procedural domain ontology, protocols and the communication language used between the agents are proposed. There is no full implementation supporting OMNI and the closest practical framework is the ALIVE toolset.

ALIVE [Lam et al., 2009] is a multi-layered framework defined by the EU funded ICT-ALIVE project[11], supporting the design, deployment and maintenance of distributed systems. ALIVE is an evolution of OMNI which uses a model-driven approach to specify the conceptual framework with metamodels. It defines three levels of abstraction: The *organisation*, the *coordination* and the *service level*. The organisational level supports an explicit representation of the organisational structure of the system. The

[11] http://ict-alive.sourceforge.net

organisational description includes roles, normative relations (e.g. permissions, obligations), and interaction scenes. The specification makes use of the OperA [Dignum, 2004] model extended with HARMON*IA* norms. The coordination level transforms the organisational representation into service-oriented workflows, specifying patterns of interaction between Semantic Web services. Workflows for agent coordination at runtime achieving organisational goals are stored. Amongst others, this level includes elements such as an ontology (containing available actions, possible goals to be achieved, the resources available in the domain, etc.) and a JSHOP2 planner producing plans to be executed by the agents. Finally, the service level supports the semantic description of services. Mechanisms for matchmaking (selection of the most appropriate service for a given task) and Web services dynamic composition are supported. A monitoring mechanism is used to track various types of runtime activities. Although there exists an ALIVE toolset fully supporting the framework, there are two main limitations: 1) The framework is focused on Service-Oriented environments: in ALIVE agents coordinate the orchestration of existing Web services; 2) Agents are not able to generate plans at runtime, but they can select precomputed plans that have been automatically built by an offline planner according to the organisation specification.

In [Hübner et al., 2011] the authors, based on primitives like norms and obligations, introduce a Normative Programming Language (NPL), that is, a language that is dedicated to the development of normative programs. They then present an NPL interpreter to compute: (1) whether some operation will bring the organisation into an inconsistent state (where inconsistency is defined by means of the specified regimentations), and (2) the current state of the obligations. They also define the Normative Organisation Programming Language (NOPL), a particular class of NPL specialised for MOISE [Hannoun et al., 2000]. Finally, they show how MOISE's organisation modelling language (with primitives such as roles, groups, and goals) can be reduced to NPL programs. While the approach provides an approach focused on monitoring norms within an execution system, no weight is given on how to choose amongst and evaluate possible operations to be executed in order to achieve goals.

Recent work detailed in [Lam et al., 2010] represents the agent organisation using Semantic Web languages. In the framework, norms (permissions, obligations, prohibitions and power) over agent actions as well as roles and role classification (a hierarchical structure between roles) are represented using OWL [Antoniou and van Harmelen, 2003] and SWRL[12]. Further to this, workflow specifications and the ontologies are made available to the agents through a centralised service which maintains the knowledge, and updates it. Workflows are graphs with states and connections (including AND/OR edges) between them and have input/output variables. They describe the tasks to be executed and the flow of control within the system. Tasks might be atomic or might be complex requiring a workflow to be executed. Agents are created dynamically by an algorithm (which takes into account the organisational norms). The input to the algorithm is a set of workflows and an ontology, and the

[12]http://www.w3.org/Submission/SWRL/

output includes a set of software agents with organisational roles and tasks associated with them, giving priority first to obligations, then to institutionalised power and permissions, and finally to permissions. Each agent then is able to start an independent process that will support the enactment of the workflow. Exceptions are handled outside the agent-view perspective and dealt with appropriately, according to whether an agent or a task has failed. While more practical, this approach follows service-oriented model and does not provide such in-depth cover of organisational elements and focuses on the coordination and workflow enactment process over the tasks to be executed.

2.2.2.3.4 Verification in Normative Multi-Agent Systems

Stemming from constantly modified contexts, norms are normally of a changing nature. As a consequence, there might occur conflicts and inconsistencies needing to be automatically detected and resolved. In other cases, possible redundancies or equivalences between constitutive normative systems might have to be identified. Although norm verification is out of the scope of this thesis, in this section we briefly describe some interesting works towards verification and/or validation of norms.

In [Vasconcelos et al., 2007] Vasconcelos et al. provide an algorithm for resolving norm conflict (e.g. when an action is simultaneously prohibited and permitted) and inconsistencies (e.g. when an action is simultaneously prohibited and obliged) in norm-regulated virtual organisations (VOs) using a unification-based technique. More specifically, they resolve the conflicts by annotating norms with sets of values their variables cannot have, thus curtailing their influence. In [Vasconcelos et al., 2009] they extend their work by adding constraints to the norms (the norms' variables) and define algorithms for resolving conflicts (overlapping values of variables, or else curtailment) by adding and removing norms.

In [Boella et al., 2008a] the authors take a previous representation of normative systems and add deadlines to the regulative norms, and research into the role of violations. They explore the way to determine if a constitutive or regulative norm is unnecessary in a normative system and if two normative systems are *equivalent*. They make the distinction between *counts-as equivalence*, *institutional equivalence*, *obligation equivalence* and *violation equivalence*.

MCMAS [Lomuscio et al., 2009] is a BDD-based symbolic model checker for the verification of epistemic and ATL (Alternating-time Temporal Logic) properties on systems described by means of variants of interpreted systems. MCMAS takes as input systems descriptions given in ISPL (Interpreted Systems Programming Language), a set of CTLK (Computational Tree Logic for Knowledge) specifications to be checked, and returns whether or not the specifications are satisfied, giving, in most cases, a counter-model if they are not. In [Lomuscio et al., 2011] the authors use MCMAS to verify contract-based service compositions. First, they define the *e-contract*, seen as

a composition of services, and all the contractually correct behaviours in WS-BPEL[13] (Web Services Business Process Execution Language). Through a compiler, they then translate the specifications into ISPL programs, supported by MCMAS, to verify the behaviours.

NormML [da Silva Figueiredo et al., 2011] is a UML-based modelling language for the specification of norms, where norms are being viewed as security policies. NormML is based on the SecureUML metamodel [Basin et al., 2005]. SecureUML provides a language for modelling *Roles*, *Permissions*, *Actions* (atomic actions such as delete, update, read, create and execute, and composite actions comprising of atomic ones), *Resources*, and *Authorisation Constraints* as well as relationships between these elements. Among other additions, the NormML metamodel extends the SecureUML metamodel with the following basic elements: *Norm*, *NormConstraint* (activation constraints), *Agent* (the agent to which the norm refers), *AgentAction* (allowed or prohibited actions) as well as *sanctions* and *rewards* for the fulfilment or violation of norms respectively. The norm specification allows to define the deontic concept (obligation or prohibition as well as and what is to be achieved) and the time period in which a norm is active, based on the execution of the actions, that is, *before*, *during* or *after* the execution of some action(s). The authors show how the formalisation allows to validate whether the norms are well-formed according to the formal language specification and to check the conflicts between these norms at design time.

2.2.2.3.5 Monitoring Normative Status

As we explained in Section 2.2.2.3.1 agents may exist within regulated environments where norms might be dynamically changing over time and, at the same time, they may be simultaneously operating in multiple normative contexts. In order to monitor the correct execution of the norms imposed, complex mechanisms which are able to interpret institutional facts as well as perceiving the normative status of the environment are needed. Such mechanisms that can interpret and follow the different phases that norms go through, from their generation, throughout their enforcement and till they get fulfilled, expired or even withdrawn, have been thoroughly examined in the literature. Although norm monitoring is out of the scope of this thesis, in this section we describe some important frameworks that cover the monitoring aspect of normative systems.

Artikis et al. view view societies as instances of normative systems [Artikis, 2003; Artikis et al., 2003, 2009] and in [Artikis, 2003; Artikis et al., 2003] they describe normative social systems in terms of power, empowerment and obligation and create operational specifications using both event calculus [Kowalski and Sergot, 1986] and the action language C+ [Giunchiglia et al., 2004], demonstrating how they can be implemented via existing tools. In his doctoral thesis Artikis [Artikis, 2003] gives examples of how roles, obligations, permissions and institutionalised power can be formally expressed

[13]http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

and uses a Contract Net protocol [Smith, 1980] to demonstrate an how the specification can be analysed and queried via an implementation in the CCalc tool [McCain, 1997]. Later, in [Artikis and Sergot, 2010] Artikis and Sergot use event calculus to formulate obligations, permissions and power and track normative states of multi-agent systems. The normative statements refer to actions rather than states to be brought about. Violations are also specified but the authors associate no specific consequences to them.

[Farrell et al., 2005] described a predicate-based event calculus approach for keeping track of normative state in contracts. Their work focuses on creating an XML-based representation of event calculus, and uses event calculus primitives to define their contracts. This however results in a very unmanageable and impracticable contract representation with a small number of norm related predicates. In addition, [Daskalopulu, 2000] showed how contract monitoring can be performed with Petri nets. However, this representation is more appropriate for contracts which can be modelled as workflows.

In [Cliffe, 2007; Cliffe et al., 2007b,a] the authors extend the formal specification of single institutions to multi-institutions. They present a top-down approach to virtual multi-institutions in which agents may reason (on-line) about and designers may analyse (off-line) external normative concepts. They introduce an action language designed for multi-institutions. The action language describes a model of the situation which can be directly mapped to an answer set program (ASP) allowing for an easy way to query properties of models. They define *institutional state* as a set of *institutional fluents* that may be held to be true at some instant. Furthermore, they separate such fluents into *domain fluents*, that depend on the institution being modelled and *normative fluents*. They define *generation functions* of *events* which have *effects* on the system and using the normative fluents they track the status of the norms at each state and detect violations.

In [Modgil et al., 2009] Modgil et al. propose an architecture for monitoring norm-governed systems. The system deploys monitors that take inputs from trusted observers, and processes them together with *Augmented Transition Network* (ATN) representations of individual norms. Hence, monitors determine the states of interest relevant to the activation, fulfilment, violation and expiration of norms. Additionally, the monitoring system is corrective in the sense that it allows norm violations to be detected and be reacted to.

Álvarez-Napagao suggests in [Álvarez-Napagao et al., 2010] the use of production systems [Davis and King, 1975] to create a normative monitor. Regulative norms are defined through activation, maintenance, deactivation, deadline conditions. The authors also include counts-as norms. The norms status can be inactive, activated, fulfilled or violated. Using a forward-chaining rule engine, they create inference rules which, by checking past events against active norms, calculate each norm's status. Norms might have several instances and the authors design the rules in a way so that

multiple instances can be handled. The formalism is then reduced to a production system (a system composed of rules, a working memory and a rule-processing engine) [Davis and King, 1975] so that it operates as a practical monitor which detects violations and enforces sanctions. The authors also provide an implementation made with the DROOLS rule engine [The JBoss Drools team, 2013]. Gómez-Sebastia et al. later extended the same monitoring framework to handle update of normative contexts (norm additions or deletions) at runtime, without needing to end the monitoring execution [Gómez-Sebastia et al., 2012]. They provided different updating processes that might take place both while taking or without taking past events into consideration.

Cranefield et al. [Cranefield and Winikoff, 2011; Cranefield et al., 2012], on the other hand, do not differentiate between norms and commitments, they instead use a generic term and study the general notion of *expectations* for future world states, events and actions. They use the Exp, Fulf and Viol operators, all of which have similar argument structure, to express a currently active, fulfilled or violated expectation. A formula $\mathsf{Exp}(\lambda,\rho,n,\phi)$ signifies that ϕ is an active expectation as a result of a rule of the form $\lambda \Rrightarrow \rho$ having fired in a (possibly prior) state specified by n. The authors also use *formula progression*[14] to show how an unfulfilled and non-violated expectation is updated from one state to the next. The approach is claimed by the authors to apply to both online and offline monitoring of rule-based conditional expectations, and their fulfillments and violations, through a model checker extended with the ability to progress expectation expressions.

Finally, Hindriks and van Riemsdijk suggest a labelling mechanism to track down the norms' status in *timed transition systems*, that is, transition systems extended with time [Hindriks and van Riemsdijk, 2013]. In their framework, norms are defined as tuples containing an activation condition, the normative target to achieve and a time before which it has to be achieved. They define *detached obligations* as obligations brought to life whenever a norm becomes applicable. On top of this, they introduce two binary relations, a blocking relation **B** where a norm can block the application of a norm appearing later and a cancelling relation **C** where a norm can cancel another norm that occurred earlier. The labelling process labels the transition system's states in two ways: 1) indicating at every state which obligations hold and 2) indicating at every state the violations that have possibly occurred. The authors then define the *detachment*, *persistence* (the continuation of activeness of a norm through time unless otherwise specified), *termination* and *violation* of norms with respect to the blocking and cancelling relations and the labels applied to the transition system. There exists no available implementation of this approach.

2.2.2.3.6 Relevant Approaches Outside the Agent Community

A widespread type of normative environment in current distributed applications is contract-based environments. For instance, the Service-Oriented Architecture (SOA)

[14] Introduced in the TLPLAN framework [Bacchus and Kabanza, 2000] by Bacchus and Kabanza.

community has shown interest in the creation of some form of (contractual) agreements as part of the specification of a distributed business process. Nevertheless this is either too abstract (focusing on XML representations capable of fully expressing *human contracts*, such as OASIS LegalXML eContracts [Leff and Meyer, 2007]) or too concrete (focusing on the specification of *service-level agreements* (SLAs) based on a few set of computer-observable parameters, also called *metrics* [Ludwig et al., 2003]). Although this thesis does not follow a service-oriented approach, in this section we mention some basic frameworks that are based on and widely used in web-based service-oriented environments.

WS-Agreement [Andrieux et al., 2007] was one of the first XML-based languages for agreements (including terms and meta information about the agreement) which also included a definition of a protocol for establishing these agreements. Although WS-Agreement is widely used, it is not possible to describe multi-party social agreements, only one-to-one contracts. It also lacks the definition of metrics in order to support flexible monitoring implementations.

Most of this has been solved in the Web Service Level Agreement (WSLA) [Ludwig et al., 2003] framework, a more expressive language targeted at defining and monitoring agreements for Web services, covering the definition of the involved parties, the service guarantees and the service definition. WSLA allows the specification of third parties and also monitoring service performance by means of metrics. However, it lacks the description of the application execution context and mechanisms for the execution of activities such as negotiating the contract. Both WS-Agreement and WSLA also lack formal semantics about behaviours and agreements (making it difficult to reason about them or to verify certain properties).

Rule Based Service Level Agreements (RBSLA) [Paschke, 2005; Paschke et al., 2005] focuses on sophisticated knowledge representation concepts for service level management (SLM) of IT services. RBSLA is an extension of RuleML including contractual norms (permissions, obligations and prohibitions), violations and exceptions. The rules are based on the logic components of Derivation, Event Condition, Event Calculus, Courteous Logic and Description Logic. RBSLA's flexibility is based on its declarative nature, allowing a more compact and intuitive representation. Monitoring of agreements is based on event-condition-action rules with proper operational semantics. RBSLA provides an expressive language for representing service-based agreements, however, it does not provide any mechanism to reason about the defined norms.

Despite their expressive power, business-oriented approaches like eBusiness eXtensible Markup Language (ebXML) [Kotok and Webber, 2001], cannot be used directly by a computational system to monitor and control the terms of the contract between software services at runtime, as this kind of contractual documents do not define how activities are to be monitored. In the case of concrete, service-level approaches monitoring is limited to the tracking of the aforementioned metrics.

Finally, the IST-CONTRACT[15] [Panagiotidi et al., 2008] project has developed a complex (syntactical and semantical) specification for the computational notation of contracts, together with configuration tools for setting up contract-based environments. The specification maps contract *clauses* to deontic statements, includes contract templates as well as instantiations of them as well as contract statements (such as whether a contract being in force), protocols for accomplishing goals related to a contract (for example notifying over a contract's fulfilment) and the context within which communication happens. Additionally, it also provides mechanisms for the verification of the clauses defined in a contract. Although there exists a toolset implementing the framework, it is only suitable for Web services applications and development is no longer active.

2.2.2.4 Agent-view: Norm-based Agents

As explained in previous sections, normative agents' behaviour is to a certain degree shaped by norms. While generally agents may be designed to comprehend and follow normative guidelines, autonomous agents go one step further, having the ability to also decide whether to adopt and whether to comply with norms, according to their own objectives and incentives. As a consequence, normative autonomous agents can adopt norms and further decide which norms to comply with (intended norms) and which norms to reject. This section analyses several frameworks that have been developed in order to deal with norms while taking the individual's perspective into consideration.

Castelfranchi et al. in [Castelfranchi et al., 2000] define a Norm Autonomous Agent (also called Deliberative Normative Agent) as an agent that is:

- Able to know that a norm exists in the society and that it is not simply a diffuse habit, or a personal request, command or expectation of one or more agents.
- Able to adopt this norm impinging on its own decisions and behaviour.
- Able to deliberatively follow that norm in the agent's behaviour.
- Able to deliberatively violate a norm in case of conflicts with other norms or, for example, with more important personal goals; of course, such an agent can also accidentally violate a norm (either because it ignores or does not recognise it, or because its behaviour does not correspond to its intentions).

The way norms are adopted and/or decided to comply with depends on the agent's architecture and its ability to perform complex reasoning processes with respect to its distinct mental attitudes. Therefore, the question that arises is how the agents should cope with these norms. Billari explains in [Billari, 2000] that norms can be introduced into the agents' decision making as either:

[15] http://ist-contract.sourceforge.net

1. Restrictions: in this case norms are reduced to constraints forcing compliance of norms.
2. Goals: in this case norms are incorporated as (prioritised) goals to be included in the reasoning cycle of the agent.
3. Obligations: norms are explicitly represented as another element of the agents reasoning cycle.

In options 1 and 2 norms are hard-coded in the agents through action restrictions or a restricted number of (accepted) goals, so agents will always behave according to these restrictions In this scenario agents do not have to know that these restrictions follow from specific norms. Therefore, the overhead in reasoning about the possible consequences of their behaviour is (almost) zero, allowing the creation of relatively simple and small agents. In option 3 norms are explicitly represented through obligations, so agents can be designed to reason about the norms. While such agents may have similar behaviour to conventional agents, they will show a greater capability when making decisions in particular situations. They are aware of the cases where violating a regulation proves rewarding. As these agents have to reason about the potential outcomes of every action, they are more complex.

In [Dignum, 1999] Dignum explains how agents reason about violating a norm. It should be noted that his proposal states that norms should not be imposed, they should only serve as a guidance of behaviour for agents, or agents will lose their autonomy. With this idea in mind, Dignum further presents an agent architecture where social norms are incorporated in the BDI cycle of the agent. In that architecture, norms are explicitly expressed in deontic logic by means of the obligation operator O and are divided in three levels:

- Convention level: norms modelling the social conventions
- Contract level: norms regarding the obligations that arise when an agent A_i commits to either perform an action α or to achieve a situation p requested by agent A_j
- Private level: the intentions of the agent are seen as commitments towards itself to perform a certain action or plan

As explained in Section 2.2.2.2, since norms in real domains are usually defined at an abstract level allowing different interpretations, it is not strange that the existing normative frameworks differ in how the norms are expressed. The representation of the norms essentially differs in the formalism which is adopted for the representation of the domain knowledge and for performing normative reasoning (for example, amongst the different approaches for implementing normative reasoners, the approaches in [Bandara et al., 2003; Craven and Sergot, 2008; Fornara and Colombetti, 2009] are based on *state transitions*). Generally speaking, most of the work on normative systems concentrates on the theoretical aspects of the normative concepts from agent societies, while little of this work focuses on developing a norm-based

system where agents are able to take normative positions into account during practical reasoning. However, practical normative reasoning in real-life systems remains an important topic in multi-agent systems. One reason for this is that normative knowledge is commonly based on formal theories such as deontic logic, dynamic logic, and so on, whose reasoning mechanisms are usually computationally very high or, sometimes computationally intractable. In the next sections we break up and discuss existing frameworks for formalising and constructing agents that use decision making with norms, focusing on the ones that apply practical mechanisms for normative reasoning.

2.2.2.4.1 Agent Frameworks Focusing on Constitutive (Counts-as) Norms

As mentioned in Section 2.2.2.2.1, norms of constitutive nature, expressed as "counts-as" statements, have been vastly explored, mainly from the legal standpoint, but the attention from logic and computing background researchers is very recent and therefore there exist few works fully exploring inclusion of constitutive norms in the agents' reasoning.

Sergot and Craven in [Sergot, 2003; Sergot and Craven, 2006] extend $C+$ [Giunchiglia et al., 2004] by adding expressions of the form α "counts_as" β (this implying that every transition of type α counts also in specified circumstances as a transition of type β), calling the extended language $(C+)^{+}$. In addition, they extend $C+$, calling the new language $(C+)^{++}$, by adding the "permitted" and "not-permitted" rules, implying in this way desired, legally permitted or not acceptable states and transitions. In this way it is possible to verify system properties that hold if all agents/system components behave in accordance with norms/social laws and to analyse system properties that hold when agents fail to comply with norms. However the extensions provide no operational semantics to work with when it comes to realistic representation of norms and practical reasoning.

In [Aştefănoaei et al., 2009] the authors extend their previous work by defining the properties' enforcement and regimentation as LTL formulas. Enforcement formulas express that a norm's violation guarantees that the corresponding sanction will be eventually satisfied, while regimentation formulas express that a norm's associated violation will never occur. They then provide a model checking component which verifies these properties. Although the approach is promising, the authors do not provide any work on reasoning and decision making over the norms.

In [Dastani et al., 2009a,b] the authors propose a programming language that allows the implementation of norm-based artefacts by providing programming constructs to represent norms and mechanisms to enforce them. The language includes counts-as rules and sanction rules, which might express sanctions in case of possible violations. A transition rule that captures the effects of performing an external action by an individual agent on both external environments and the normative state of the multi-agent system is defined. The process is as follows. First, the effect of the action is computed.

Then, the updated environment is used to determine the new normative state of the system by applying all counts-as rules to the new state of the external environments. Finally, possible sanctions join the new environment state after the application of the relevant sanction rules to the new normative state of the system. The execution of a multi-agent program is considered as interleaved executions of the involved agents' behaviours started at the initial state. The authors also suggest a sound and complete logic which can be used for verifying properties of the multi-agent systems with norms implemented in the proposed programming language at some states reachable by an execution path. Although promising, the approach does not suggest a way of reasoning over such possible execution paths and does not evaluate their social or private utility, assuming that the actions to be taken are decided by some external mechanism.

2.2.2.4.2 BDI-based Normative Autonomous Agents

The BDI model (see Section 2.1.1.1) is by many now considered the standard approach to agent architectures, as it provides a wide span of behaviours from solely deliberative to simply reactive, depending on the agent cycle and plans' implementation. Given the agent's autonomous nature, several researchers have made important attempts to incorporate social influences into BDI agents, expressed not as rigid constraints, but instead, as norms, aiming for an even larger spectrum of behaviours to be represented in one sole framework.

Dignum et al. in [Dignum et al., 2000] propose a modification of the BDI architecture into a socially motivated deliberation process, taking the influence of social obligations and norms on the deliberation process into account. The approach focuses on the process of generating (candidate) intentions from normative influences. The authors use the notion of both *norms* and *obligations*. Norms have a social aspect and make cooperation and coordination and interaction more efficient. Obligations on the other hand are associated with specific enforcement strategies which involve punishment of violators and in this way they restrict autonomy. For norms, the preference ordering is related to the "social benefit" attached to different worlds and for obligations, the preference ordering is related to "penalties" imposed for violation. The basic deliberation cycle, which includes a process of events selection and plan generation through the selected events, is modified by involving the notion of *deontic events* and *potential deontic events*. The former are generated from changes in the norms, obligations, and beliefs of an agent. To respond to these deontic events, each agent has plans whose invocation conditions are deontic events. The latter are events that may also exist, depending upon what *plan-options* (decided by the *option-generator*) are decided in the deliberation step. An additional step is then introduced to the agent's loop is as follows: some set of events is chosen and is augmented with potential deontic events that are generated by repeatedly applying the introspective norms and obligations. This augmented set of events is the one that will be used to determine the plan-options

calculated. The deliberation step then selects between these sets of plans on the basis of the preferences. There is no implementation available for this work.

Broersen et al. [Broersen et al., 2001] present the BOID (Belief-Obligation-Intention-Desire) architecture as a model of a norm-governed agent. It contains four components (*B*, *O*, *I* and *D*) where B stands for beliefs, O stands for obligations (representing commitments towards social rationality), I stands for intentions and D for desires. The behaviour of each component is defined by formulas. More specifically, *extensions* are propositional logical formulas defining each component's behaviour in the form of defeasible rules. An *ordering function* on rules is used to resolve conflicts between components. The authors propose a *calculation scheme* to build in each cycle the new set of logical formulas. Then, in order to produce the whole extension of the agent every time, the process starts with the observations and calculates (through the calculation scheme) a belief extension and then, when done, applicable rules from O, I and D are applied successively, each time feeding back the belief component for reconsideration. The order in which components are chosen for rule selection determines the kind of character the agent possesses. For example, if obligations are considered before desires, the agent is regarded as a social agent. One drawback is that the creators only consider extensions in which the belief component overrules any other modality. Furthermore, the ordering function is fixed for each agent.

In [Meneguzzi and Luck, 2009b] the authors extend a BDI agent language, enabling the agents to enact behaviour modification at runtime in response to newly accepted norms. According to their specification, a norm (obligation or prohibition) can refer to a state or an action and has a validity period defined by an activation and an expiration condition. An agent might accept or reject a norm (a process not dealt with inside the framework). The authors provide methodologies to react to norms' activation and norm's compliance. These consist of forming new plans (inserting them to the plan library) to comply with obligations and preventing existing plans (deleting them from the plan library) that violate prohibitions from being carried out. They demonstrate their framework's practical usefulness via an implementation in AgentSpeak(L) [Rao, 1996].

Another interesting piece of research on normative BDI agents is [Criado et al., 2010a,b]. They base their work on graded BDI agents. According to the graded BDI architecture (n-BDI for short) [Casali, 2008], an agent is defined by a set of interconnected *contexts* (mental, functional and normative contexts), where each of them has its own logic (i.e. its own language, axioms and inference rules). In addition, *bridge rules*, whose premises and conclusions are in different contexts, are inference rules derived by one context and modifying the theory of another. In [Criado et al., 2010a,b] the authors propose an extension of the n-BDI architecture, in order to help agents to take practical autonomous decisions with respect to the existence of norms. They use rules to decide on norm adoption as desires. They apply deliberative coherence and consistency theory for determining which norms are more coherent with respect to the agent's mental state. The authors suggest a methodology to detect and resolve inconsistencies

between norms and desires. A basic difference in their approach, however, is that it mainly focuses on the reasoning over the adoption (or not) of instantiations of norms rather than suggesting whether and by what means to achieve the norms' fulfilment. Moreover, they assume: 1) a quantification of the mental context (beliefs, desires, intentions) associated to the certainty degree of each of these elements; 2) punishing and rewarding reactions for each norm's violation or fulfilment; 3) predefined functions that will determine the adoption or not of a norm instantiation; 4) predefined values for the weights expressing the strength of the mental context elements that are related to a norm. We find that it can be difficult to estimate or predetermine all these factors when designing norms. One more limitation is that the approach only considers obligation norms.

In [Ranathunga et al., 2012] Ranathunga et al. try to see norm monitoring from the individual agent perspective rather than from the standard organisational perspective and integrate their previous theoretical work [Cranefield and Winikoff, 2011] on expectation monitoring (explained in Section 2.2.2.3.5) into the Jason [Bordini and Hübner, 2006] platform. They extend the Jason interpreter by adding two internal actions that represent the initiation and termination mechanism for the monitor to its standard actions library. They define the extended Jason configuration to be a combinatory configuration of the Jason agent and the monitor. Whenever the monitor detects fulfilment of violation of a rule it notifies the agent, which in its turn can react accordingly by executing predefined plans triggered by such events. The authors explain how a current limitation of the system only permits the handling of only one rule at a time.

In [Alechina et al., 2012] the authors use a norm formalism for obligations and prohibitions that contains pre-specified sanctions in case of violation. Based on the 2APL agent programming language, they extend its PG-rules reasoning rules (rules that select pre-defined plans to be executed) to contain event-based rules that initiate norms (obligations and prohibitions) and name it N-2APL. Whenever these event-based rules are triggered, obligations are adopted as goals and prohibitions are activated. By defining a priority ordering function that indicates the agent's preferences over goals as well as sanctions for violating obligations and prohibitions, they design an algorithm to calculate the set of plans that will be optimal with respect to this function. An agent might have a "social" character if its obligations are preferred to its goals, trying to primarily fulfil these obligations. On the other hand, if an agent gives priority over its goals, then it might end up breaching a lot of norms and getting highly sanctioned for this.

Taking Alechina's work as basis, in [Dybalova et al., 2013] the authors et al. make an integration of N-2APL with the organisation programming language 2OPL [Dastani et al., 2009a] in order to create a system where, instead of the norms being an internal part of the agent, they are imposed exogenously, that is, by a normative organisation. The communication between the 2APL agents and the 2OPL normative organisation is done through a tuple space. These tuples are accessed just like an external environment and they represent the state of the multi-agent system and its normative state

in terms of active obligations, prohibitions and applicable sanctions. As we explain again later in Section 3.3.2, the methodology uses the plan library (pre-stated planning rules) of the agent and does not explore a dynamic planning mechanism for the creation of new plans.

2.2.2.4.3 Rule-Based Normative Agents

There exist norm formalisation frameworks that are founded on rule-based languages, in which norm triggering is determined by event-driven rules. Two approaches of special interest to us are the ones made by García-Camino et al. and by Fornara and Colombetti.

In [García-Camino et al., 2009] the authors introduce a rule-based language to capture and manage the *normative positions* of the agents (permissions, prohibitions and obligations). The language defines a standard production (forward-chaining) system enhanced with constraint satisfaction techniques and makes it possible to check fulfilment and/or sanction un-fulfilled normative positions, i.e. obligations, permissions and prohibitions, that are bounded with constraints. Its rules correspond an existing state of affairs to a new one, modelling in this way transitions between states of affairs. At every state an exhaustive application of rules to the modified state is performed (adding or removing formulas that represent agent-related events), yielding a new state of affairs. The authors' rule-based formalism additionally includes constraints allowing to supplement (and apply) electronic institutions with norms. They depict their work by representing and enacting well known protocols via institutional rules. However, due to the forward-chaining calculation, it lacks the ability to plan (i.e. determine the succession of sets of events that need to take place in order to achieve a given state of affairs from a given initial state) and also makes it impossible to postdict (i.e. determine the previously unknown facts in a partial initial state given a final state and the sequence of sets of events that have occurred).

In the framework described by [Fornara and Colombetti, 2009], the authors express norms in terms of occurrences of events and actions. Norms can be obligations or prohibitions and are expressed as commitments to perform or refrain from performing an action within a specified time window. The authors use Event-Condition-Action (ECA) rules (of the type "On *Event* If *Condition* Do *Action*") to generate commitments with operational semantics from the norm specification. The lifecycle of commitments is composed of different norm states, namely unset, pending, fulfilled, violated, cancelled, extinguished and irrecoverable. In this approach, agents are expected to react to commitment violations through pre-specified sanctions. The suggestion of Fornara and Colombetti mainly deals with under which conditions (events) and how norm constructs are created. Norms can be used to observe whether the agents' behaviour is compliant with the specifications and able to suitably react to violations and the authors suggest no reasoning mechanism on the basis of the sanctions/rewards

In [Vanhee et al., 2011] the authors provide an overview of the issues encountered in implementing different norm aspects in agents. Using a simulation scenario they implement rules that lead to the adoption or not of a norm, the generation of norm compliant plans and the monitoring of norm enforcement. However they provide hard coded mechanisms for dealing with norms (norms are integrated into the agent's code) and as a result several implementation complications occur.

Finally, van Riemsdijk et al. in [van Riemsdijk et al., 2013] explore a formalism where norms are defined as temporal LTL formulas over actions (***if*** *condition* ***then*** *some action has to take place* ***before*** *some other action*). They translate these formulas to *Separated Normal Form (SNF)* [Fisher, 1997] formulas which have an operational functionality in computational systems. They show how these can be handled appropriately at every computational step to block the execution of actions that are forbidden by a norm or to trigger the execution of actions that are required by the norms.

2.2.2.4.4 Rational Agents Normative Reasoning with Uncertainty

In Artificial Intelligence literature (mainly in Robotics), there exists a long line of work towards agent reasoning in environments where the outcomes of the agents' actions are not fully known, therefore containing some level of uncertainty. Nonetheless, norms (or their influence) have not been explored until recently.

Fagundes et al. [Fagundes et al., 2010, 2012a,b] use probabilistic environments. In specific, they extend the Markov Decision Process (MDP) [Bellman, 1957; Puterman, 1994] to include a set of norms, calling it NMDP. They define contracts to be agreements between two or more agents, containing a set of norms. The authors distinguish between two kinds of norms, *substantive* (which provide guidelines as to how agents should behave) and *procedural* (specifying reactions on violations of the substantive norms). Substantive norms can be obligations or prohibitions and their so called normative content specifies a state that needs to be achieved or avoided respectively. Sanctions contain a set of modifications to the domain state-transition function and the agent capability function that are applied in case of a violation. The authors then create an algorithm that constructs the state space that matches the normative content of the norms and searches through it, identifying violations (if no state achieving an obligation or if a state that fulfils the normative content of a prohibition is visited). For each violation, the algorithm calls a function that represents the respective sanction and applies the respective updates. They then generalise this methodology for contracts to estimate risks for contracts comprising of sets of norms. They show that by repetitively applying this algorithm over a set of norms (a contract) and comparing the expected utilities associated with the initial state of each MDP, picking the one that maximises the expected profit, they can decide which will be the most profitable contract to sign. We find this approach towards norm-aware probabilistic reasoning inspiring, as it allows the state space to be explored allowing norms to be potentially violated or not and weighing over which is the best norm set compliance with respect

to the probabilistic environment to provide the best overall outcome. However, we detect two weaknesses: 1) NMDP needs a special implementation of the MDP process and algorithm to be able to handle the norm specification; 2) Norms specify modifications to an agent's capability functions and the state transition model. We find this somewhat counterintuitive (compared to separate norm and domain representation layers) and difficult to handle since it adds a lot of extra work to the agent's designer. Additionally, considering a multi-agent environment, such adaptations should possibly be specified for each agent separately, complicating the task even more.

In [Oh et al., 2011] the authors present a proactive planning agent framework that predicts the user's probable normative violations, permitting the agent to plan and take remedial actions before a violation actually takes place. The authors employ a probabilistic plan recognition technique to forecast a user's plan for his future activities. More specifically, given the agent's planning domain (state space), the state transition probability depends only on the current state, and the user's action selection on the current state and the specific goal. Consequently, by applying the chain rule, they calculate the conditional probability of observing the sequence of states and actions given a goal. The norm reasoner then traverses each node in the plan-tree and evaluates the related user state for any norm violations. In case of violation occurrences, it generates a set of new goals (or tasks) for the agent by finding near states that are compliant with all norms, preventing in this way those norm violations from happening. As the user's environment gets modified, the agent's forecast is constantly updated, and therefore the agent's plan to fulfil its goals is regularly revised during execution. In this piece of work, the authors' goal is not to guide the user in finding optimal planning solutions, but instead, to support the user's decision making process by identifying and making amends for the plan's weaknesses.

2.2.2.4.5 Normative Autonomous Agents Using Planning

In recent years there is a trend towards analysing and exploring the agent's possible future execution paths and deciding through different evaluation mechanisms which path is the best, with respect to pre-determined properties. This is done either through classical planning domains or HTN domain representations. It has to be noted that in most cases, plans are precomputed and available in some sort of *plan library* for evaluation. Some other frameworks use existing BDI frameworks, where agent plans are represented through complex execution rules, and modify the agent's language interpreter and lifecycle to include and process normative rules (such approaches have been described in Section 2.2.2.4.2). The rest of this section details frameworks that directly process and evaluate plans through planning mechanisms.

In [Oren et al., 2011] the authors use a mechanism to choose a plan that will achieve individual and global goals while attempting to abide by a set of norms. They represent the environment affecting the agent as a transition system and the plans as Hierarchical Task Networks (HTN) [Ghallab et al., 2004] with the nodes specifying the

actions that take place. They make use of a rule language to specify normative rules that identify the cases in which a norm starts, and ceases, to exist. Additionally, they adopt a utility based model of norm compliance. More specifically, they make the assumption that the execution of a plan results in some base utility, and that different types of norms are associated with different utility measures. They then create an algorithm that selects a path through the plan, and a set of norms (created by the rules as actions are executed) with which to comply, that is conflict free, and which leads to maximal utility. Conflicts are resolved by selecting actions where the cost of violating one set of norms is outweighed by the reward obtained in complying with another.

In [Oren and Meneguzzi, 2013] Oren and Meneguzzi develop a norm derivation mechanism which operates by analysing precomputed plans available to the agents. The framework assumes that agents in the environment share a static plan library that contains plans consisting of subtasks, generated offline by a Hierarchical Task Network planner. The authors consider two types of (conditional) norms, obligations and prohibitions. They design an algorithm to 'guess' possible norms being followed by the agents. The algorithm is based on plan recognition and implements an observation technique where, having knowledge of the plan library, one can analyse a plan and the sequences of its actions and contrast it against others to deduce goals that the agent pursues. These goals then indicate obligations and violations followed by the agents. Since this unrealistically assumes that agents always comply with the norms, the authors take it one step further and extend the algorithm to learn from possibly violating agents too. The advanced algorithm keeps counters of possible obligation or prohibition existences and in case a pre-specified threshold is exceeded then they are assumed to exist.

2.2.2.4.6 Plan Labelling Frameworks

Some normative frameworks focus on comparing or labelling executional paths with respect to a (set of) norm(s). In the rest of this section, we analyse two of these.

In his PhD thesis [Kollingbaum, 2005], Kollingbaum presents the NoA system, comprising the NoA language for the representation of plans, norms and contracts, and the NoA architecture, which acts as an interpreter and executor of these specifications. The plans get instantiated at runtime according to whether they can satisfy a norm and they are labelled as *consistent* or *inconsistent* with respect to the currently activated norms. With this labelling process, the deliberation mechanism gets informed about potential norm violations. While the semantics used for the specification of the norms and a norm's activation and deactivation as well as plans specification are well structured, the approach has the disadvantage that it uses sets of predefined plans in order to achieve tasks or states. Although the authors claim that there exists an implementation of their NoA agents, we have been unable to access it.

Craven and Sergot explain in [Craven and Sergot, 2008] how, from the normative semantics point of view, one can "label" a transition system representing the agent's

actions providing normative semantics over what should and should not occur. They extend the language $C+$ with two types of rules and call it $(C+)^{++}$ (alternatively $nC+$) [Sergot and Craven, 2006]. These rules can be *state permission laws* of type "n: not-permitted F if G" and *action permission laws* of type "n: not-permitted α if ψ". They then colour the transition system states according to those kinds of laws. *Green* states simply represent "acceptable" states whereas *red* states represent "unwanted" states. Also one can see that the existence of a *red* transition (an unacceptable action) in a plan, means that this plan is violating some norms. Given this formalism they describe various types of agent behaviour.

2.2.2.4.7 Action Language and Abductive-Based Approaches

Several approaches focus on enriching existing action representation formalisms and languages with normative elements. Since one of the main challenges in the implementation of practical normative reasoning concerns the modelling of dynamic domains in which information may be incomplete and/or inconsistent, most of these attempts concentrate on how reasoning about actions is done and specifically on the design of languages based on action effect axioms as well as drawing inferences from the axioms.

In [Panagiotidi et al., 2009], an extension of action language A [Gelfond and Lifschitz, 1998] is presented in order to allow modelling norms in dynamic domains based on Answer Set Programming (ASP) semantics. In this, norms have activating, deactivating and maintenance condition and the authors specify properties of a norm's lifecycle such as active, inactive and violated. They additionally provide planning rules and foresee which of the defined properties hold in future paths of a maximum length.

In a different approach, in [Gelfond and Lobo, 2008] the authors come up with an extension of language A that represents authorisations and obligation policies in dynamic environments. They additionally provide techniques for verifying compliance of performed actions within the specified policies. However, the work lacks mechanisms for reasoning as well as prediction for future desirable and undesirable behaviour.

The $\mathcal{S}$CIFF framework explained in [Alberti et al., 2008] consists of a specification language for agent protocols and a proof procedure (based on abductive logic programming) to prove properties of protocols. Their positive and negative expectations (of events in a protocol) can be understood as, respectively, obligations and prohibitions. However, that work deals with proving properties of protocols, rather than studying how the norms affect the agent's reasoning.

2.3 Summary

In this chapter we have analysed related work on (practical) reasoning agent frameworks, norm-governed agent systems and a specific type of normative systems, that is contractual systems, widely applied in service-oriented architectures. As seen in Sections 2.2.2.4.1-2.2.2.4.4 there have been many proposals to implement normative reasoning through various mechanisms. Additionally to these, there exist some frameworks implementing normative reasoning, based on model-checking techniques (i.e. [Lomuscio et al., 2009], explained in Section 2.2.2.3.4), theorem provers, transition states, meta-interpreters in logic programming (i.e. in [Artosi et al., 1994; Antoniou et al., 2008]) and/or even action language extensions.

Nevertheless, despite the large amount of theoretical work on normative agents, there are still very few implementations offering practical reasoning within an environment where norms act as guidelines for the agents. Therefore this motivates our interest in formalising and creating a framework where agents are able to reason and make decisions based on normative guidelines. In Chapter 3 we will describe the kind of norm language that is needed to specify normative environments in a way that normative agents can properly reason about norms. We will see that it should cover several levels of communication and will define its operational semantics. Then in Chapters 4 and 5 we will extend the norm semantics and then present a practical architecture that can reason with these norms.

Chapter 3

Conceptual Framework and Architecture

In Chapters 1 and 2 emphasis has been given to normative systems, that is, systems the expected behaviour of which is based on a set of regulations. This chapter presents our proposal of a normative framework, its architecture and the formalisation of the normative elements involved. The detailed framework and architecture will provide the context in which the practical normative reasoning mechanisms (presented in Chapters 4 and 5) will work and the context where normative agents (presented in Chapter 5) will operate.

It is useful to distinguish between the framework and the architecture. The framework is a theoretical specification of a system functioning within a normative context. It is helpful in providing an accurate understanding of the system behaviour and establishes a common conceptual basis to be shared between agents in different contexts. A well specified framework allows for interoperability between different applications, as well as providing a semantics for norm interpretation and verification. Such semantics allow one to unambiguously describe the state of an application as it executes.

The architecture specifies how a norm based system should be implemented and is naturally built on top of concepts defined by the framework. As a consequence, the architecture can be seen as an instantiation of the normative aspects of the framework: a set of middleware and design patterns to support management of norm-aware software agents.

Section 3.1 provides the requirements analysis of the normative reasoner to be designed and implemented and in Section 3.2 the general framework is presented. In Section 3.3 we provide the architecture of a norm-aware agent. In Section 3.4 we talk over some issues to be taken into account when modelling norms and how we deal with them in our framework. Finally, in Section 3.5 we provide a short discussion on the framework elements presented through the chapter.

3.1 Requirements Analysis

Based on the problem statement presented in Chapter 1 we have performed a thorough software requirements analysis in order to achieve our objectives. Firstly because it leads to better understanding of the design needs but also because it provides a detailed description of the behaviour of the system to be developed. This section presents a complete and focused list of requirements that the practical agent reasoning framework and its implementation have or should incorporate. Requirements are marked in the fashion *[R*.*]* to ease referencing.

3.1.1 Functional Requirements

In this section we present the functional requirements for our framework. These concern the specifications of the software systems and the tools to be used. Functional requirements also specify how the framework should behave and what functionalities it will be designed to provide.

3.1.1.1 Agent Model

This section introduces the agent model requirements. These might include the agent architecture design decisions over the particular agent components.

R1.1 **Deliberative, means-end, norm-oriented reasoning mechanism**
The adopted normative standards must *influence the agents' practical reasoning* while operating within a complex environment, resulting in a highly sophisticated, norm-driven agent. Therefore, the framework should implement a reasoning mechanism that takes into account the norms available to or imposed on the agent. This reasoner should aid the agent to reach its objectives, by deliberating on its knowledge over the environment and considering how and when to conform to the organisational norms, keeping the agent's interests and preferences in mind.

R1.2 **Decision making process guided by user preferences**
The agents must be endowed with the *ability to take user's preferences and other weighting factors into consideration*. Therefore, the decision making process must be flexible enough to handle multiple and diverse factors (e.g. costs, weights, evaluation criteria) related to or reflecting the users' preferences.

R1.3 **Goal driven decision making**
The system should model and produce an *autonomous, goal-driven agent*, that is able to take the environment's normative influence into account in its decision making. Therefore, the framework should support decision making towards one or more specific goals. The goals formulate a specific state of affairs to be achieved by the agent.

R1.4 **Agent capabilities specification accommodated by framework**
We expect our system to create a decision making mechanism for goal-driven agents living in a normative environment. These agents evaluate the environment's influence and adjust their behaviour according to its normative guidelines. This adjustment might result in the re-consideration and creation of plans that reach their objectives. Therefore, the framework should include an expressive and interoperable representation capable of capturing a wide variety of agent behaviours. This will include operational knowledge and semantics to the operations and actions (effects and side effects) available or relevant to agents. For this reason, a precise and operational action description language describing under what circumstances actions can be performed by agents and their effects on the environment is necessary.

R1.5 **Adjust in case of relevant environment change**
The framework should provide a flexible reasoning mechanism where agents take the environment's normative influence into account. We expect that the environment will be constantly submitted to alterations, as internal and external influences dynamically modify its status. Therefore, the agent's reasoning cycle must be able to adapt in case of (possibly unexpected) event-triggered changes. The adaptation should include a reconsideration of the agent's beliefs and objectives, as well as the activity planned to reach these objectives.

R1.6 **Norm conflict toleration**
The agent must be able to *detect conflicts* that might occur while operating in a complex normative context and should be able to *invent strategies that possibly resolve them*. Therefore, the framework should be able to cope with environments that include conflicting norms. In case a conflict results in a dead end, the framework should be able to detect this at least at execution time. Otherwise, it should be able to consider the benefit and loss for each possible outcome (considering, for example, possible violations or non-satisfiability of norms) and come up with the most profitable for the agent solution.

3.1.1.2 Domain Model

This section analyses the requirements that our reasoner should comply with with respect to the representation of the world. Since agents operate within social systems where complex knowledge models might exist, our system should be able to use and comprehend such elaborated models.

R2.1 **Full domain/environment definition**
We desire to build a framework where agents might operate in and interact with a shared normative environment. Agents should have a sufficient knowledge about the domain in which they are expected to act and should be able to often update their information from observations of occurring events. The framework therefore should include a wide definition of knowledge representation on various levels, one that allows for flexible and powerful world representation. Since it is not always possible for an agent to have full awareness of the world, the agent should be able to have implicit access to that kind of knowledge (possibly stored somewhere). Specifically, we aim for representation for: *a*) Ontology: The knowledge representation allows for the use of ontologies when handling domain knowledge, and allows as well for extra knowledge about the world model *b*) Conceptual model abstracting the representation of agent's capabilities, goals, state of affairs *c*) Operational semantics for domain knowledge elements

3.1.1.3 Norm Model

In this section we explain what the requirements are for handling the norms. Since our objective is to design a reasoner that operates within a normative system, norms are a crucial element in the design of the framework. We detail how norms are to

be modelled and the attributes that the system should maintain in order to make decisions with respect to a normative setting.

R3.1 **A well defined normative model allowing the clear and unambiguous interpretation of norms on a operational level**

Our framework should provide a mechanism for agents to reason within normative environments. This implies a clear comprehension and undoubted interpretation of the norms on an implementation level. Therefore, the framework should include a well defined norm model which reflects the formal patterns of relationships between agents and the behavioural patterns that dictate their behaviour and activities.

a) Norm definition: The concepts of obligation, permission, prohibition, and violation should be defined in the norm model of the language, in a logic that allows for an unambiguous and verifiable declaration of regulations.

b) Operational norm specification: Norms should be translatable into specific rules, violations and sanctions that can be effectively used by a computational system at execution time within the framework.

c) Norm lifecycle: The framework defines the lifecycle of norms (norm activation, norm violation, norm fulfilment etc.).

d) Event-triggered norms: The language should allow for norms that come into force on being triggered by conditions or events. That points to the norms having a conditional aspect that must be captured through appropriate semantics.

e) Support for complex norm representation and dependencies: The framework and the norm formalisation should support simultaneous norms, the effectiveness of which (possibly) depends on the status of other norms.

R3.2 **Mechanisms for agent behaviour monitoring of norms**

Our objective is to provide a generic mechanism to support normative reasoning. This will be used by agents to create and evaluate their plans aiming to achieve their goals. This means that the framework should allow for the monitoring of all agents' behaviour by keeping track of their actions, etc. Additionally, it might provide mechanisms to monitor the norms by keeping track of their fulfilment, violation, etc. What is more, the agents should be able to invent solutions to overcome possible conflicts or inconsistencies between the deontic restrictions.

3.1.2 Non-Functional Requirements

In this section we specify non-behavioural criteria that will be essential to establish the smooth operation of our reasoner. These mainly concern the reusability, extensibility and performance of the system.

R4.1 **Agent-oriented architecture**
The system should model and produce an autonomous, goal-driven agent that is able to take the environment's normative influence into account in its decision making. Therefore, the system architecture should follow an agent-oriented model. Each agent will have the reasoning mechanism incorporated and perform it in order to operate within an agent society.

R4.2 **Open standards support**
We aim to provide *a generic mechanism* to support *normative reasoning* that can be used by *agents* to *create and evaluate their plans* in real time. Therefore, the architecture of the framework should be based on a set of open standards that have a strong support from artificial intelligence and agent communities.

R4.3 **System platform-independent model**
The reasoning framework should be modelled in such a way that is independent of the technological platform (operating system, programming language, software libraries, etc.) used to implement it.

R4.4 **Strong focus on semantics at domain, agent context, ontology, normative level**
We envision agents that dynamically reason over norms imposed by the system while operating in non-static, complex environments where possibly multiple layers of knowledge exist and are shared between parties. Proper handling of this knowledge and communication between entities within such a system is essential. Therefore, the framework should put strong emphasis on semantics at all levels, that is domain, agent context, ontology and normative levels. All formalisations should consist of concrete and operational theories that provide semantics for the elements involved. A model capturing the environment within which the agent operates and any external changes should exist.

R4.5 **Tool Support for norm and domain representation**

A motivational force for this thesis is that few existing frameworks provide a *functional representation of normative concepts* and therefore flexibility in terms of norm adoption and reasoning about courses of action. In our normative reasoner, the agents must be endowed with the *ability to take user's preferences and other weighing factors into consideration* as well as *dynamically become aware of new norms*. Therefore, we require practical, implemented tools that will be able to facilitate the user when modelling the knowledge necessary for the agent reasoner to operate. Examples of such tools are:

a) A practical tool to represent norms on an abstract level

b) A practical tool to represent domain knowledge

R4.6 **Support for multiple standards and extensibility**

We expect our framework to have the ability to perform in dynamic and complex domains, being modified by possible interaction with third parties or by external events that are not caused by the agent. It is not always realistic to expect to know beforehand in what ways the normative reasoner could be integrated in or connected to other components providing different functionalities (for example different types of BDI agent implementations). The reasoning framework therefore be generic enough and extensible, allowing different knowledge representations and agent frameworks to make use of it. Further to this, the norms language should facilitate the handling and conversion to other standards at runtime.

R4.7 **Soft real time performance**

The work in this thesis aims to provide a mechanism to support normative reasoning that can be used by agents to create and evaluate their plans in real time. Therefore, ideally, the reasoning should be taking place in real time, as it would happen in real life. Nevertheless, this might not be a hard constraint, as few applications can actually guarantee the absolute compliance to this need. Therefore, instead of requiring *hard real time performance*, where the system is absolutely bound to meet all deadlines, we assume the need for a *soft real time performance* [Liu and Layland, 1973]. The latter refers to a more tolerant approach where, the more the system fails to give results within the time constraints, the more its quality of service falls. Still, allowing such flexibility in the agent's functioning means that such performance failures will not result in a complete breakdown, but might be handled appropriately by alternative reasoning mechanisms, precomputed or even random sequences of steps to be followed.

R4.8 **Reasoner's response time priority over optimality**
As we wish to design a mechanism that creates and assesses the agents' plans related to real-world domain representations and problems reflecting real-world situations, we would strongly desire this to happen in real time. However (as explained in R4.7), this cannot be achieved in all cases. We compromise our system's optimality by giving emphasis on the response time. That means that we allow the reasoning mechanism to opt for plans that might not provide optimal solutions when it comes to norm compliance and the agent's overall benefit, but give a valid result within the desired time limit.

R4.9 **Validity**
The system must provide a mechanism to support normative reasoning that can be used by agents to *create and evaluate their plans* aiming to achieve their goals. Therefore, the system should produce a valid and executable (with respect to the agent's capabilities) series of actions to be followed by the agent.

3.1.3 Technical Decisions

Having discussed the requirements for the practical normative reasoner we proceed with explaining the approach taken towards designing and implementing such a framework. Several technical decisions and technology choices based on the requirements stated in Section 3.1 have been made in order to work towards creating the normative reasoner. The following capture and justify our main decisions towards the creation of the practical normative reasoner:

- We use an organisational normative model to capture basic notions [derived from R3.1, R4.6]. As explained in Section 2.2.1 organisations provide a framework for the representation of entities, interactions between them and possibly structural and functional aspects of a system. Since our agents live and operate within societies, an organisational model might provide useful structural and notional representations such as agent, role, norm etc., within our agent community. We define our organisational elements by developing a conceptual framework in Section 3.2.
- We use complex norm semantics based on extended deontic logics to represent norms [derived from R1.6, R4.4]. The idea behind this is that Standard Deontic Logic is too abstract to capture the operational essence of a system that contains norms. For this reason, we elaborate semantics that extend existing deontic logics and add practical meaning to the norms' potential status (e.g. under exactly what conditions a violation occurs, or, how exactly a norm's fulfilment gets accomplished), in order to be able to implement them in a reasoning system. These semantics are developed and explained later in the thesis, in Chapters 4 and 5.

- We adopt a Model-Driven Engineering (MDE) approach [derived from R4.2, R4.3]. Model-Driven Engineering (MDE) is a methodology and technology aiming to alleviate the platform dependency occurring in complex systems. In model-driven design, models are the primary element used to develop an application. MDE captures the elements of a system through *metamodels* and enables the separation of the conceptual and the implementation level including low-level format specifications for the system's inputs and outputs. In this way high level platform-independent models are constructed expressing the design of the system in an abstract way and leaving out programming and formatting details, while at the same time allowing for transformation engines and generators. Our choice provides many advantages when dealing with the development of a normative reasoner: a) it provides a clear, abstract representation of the norms and other perceptional elements (such as capabilities, environmental information) available to the agents, b) allows easy importing and exporting other models and ontological elements into an existing model, c) through high-level norm abstractions, MDE facilitates dynamic norm interpretation by multiple target platforms and allows to easily define transformations of the agents' perceptional elements among source and target languages to automate the development and integration of the norms within the reasoning process of an agent and d) it allows analysis and consistency checking of norm properties independent to the language used to model them.
- We perform practical reasoning with a planner [derived from R1.1, R1.3, R1.4, R2.1, R4.7, R4.8, R4.9]. Despite the high complexity proven for the general case of planning problems, recent advances in planning research have led to the creation of planning algorithms that perform significantly better than previous approaches to solving various problem classes [Weld, 1999; Likhachev et al., 2005]. Most current planning algorithms make use of different techniques, either combined or separately: 1) expansion and forward-searching[1] in a planning graph [Blum and Furst, 1997] and 2) application of and alternation between heuristic methods. Such improvement in the efficiency of modern planners makes it possible for them to be included to an agent's deliberation cycle within soft real time [requirement R4.7]. The benefits of adopting such an approach are that existing software allows for domain definitions in a simple action language and that most of the extensions nowadays support features such as path costs, preferences and constraints and even investigating temporal properties over paths. We search for and select a planner that matches the criteria and accommodates appropriate semantics for the purpose of normative reasoning[2].
  - Select a planner able to manage action descriptions. An agent has specific capabilities that might be performed under certain circumstances. Such a specification must be dealt with by the planner. In a planning domain, the

[1] Further to this, some planners compile the planning problem into a logical formula to be tested for satisfiability (SAT) [Kautz and Selman, 1992].

[2] In this thesis we use two planners. A planner that uses search control rules, TLPLAN in Chapter 4 and a PDDL 2.1 planner, Metric-FF in Chapter 5.

most common way of representing the power to perform tasks is through action specifications.

  – Use a linear planner. Linear planning tries to solve a problem by using stacks instead of sets of goals. That means that the planner works solely on one goal before moving to the next one. While non-linear planning provides sound, complete and optimal results with respect to the plan length, the complex algorithms required and the huge search space makes it very difficult to have an efficient implementation. For this reason, we choose to work with linear planners, since, although they provide suboptimal results, there exist several fast and fully functional implementations.

- We adopt a BDI agent architecture [Bratman, 1987]. The reason for this is that the BDI architecture provides a well-known and widely accepted operational and reasoning framework (including various practical implementations) based on mental attitudes (beliefs, desires and intentions). In our case, the mental attitudes reflect and can be easily mapped to our problem elements (e.g. agent capabilities, goals). Additionally, BDI provides a reasoning lifecycle which can integrate a planning mechanism as part of the reasoning process (the means-ends reasoning step). [derived from R1.1, R1.3, R1.4, R1.5, R4.1, R4.2].
- We use the 2APL agent framework [Dastani, 2008]. The reason for this is that it provides in a clear and simple syntactical representation for the main elements of the BDI architecture (beliefs, goals and plans) and makes it easy to transform beliefs and goals to planning domains. [derived from R1.4, R1.5, R2.1, R3.2, R4.5, R4.6].

3.2 Conceptual Framework

As explained in Section 3.1.3, in order to effectively introduce the independence of the norm representation and the final norm reasoner within our agents, we have chosen an MDE approach. We have created a conceptual framework which defines all the elements needed in our architecture. In this section we will deal with these analytically.

The conceptual framework and the platform-independent norm representation is specified through abstract metamodels for the initial state, the actions, the plans and the norms. At execution time metamodel instantiations (called models) can be created, visualised by means of tools and used by our reasoning architecture. As we will see in Section 3.3, we also support various automated transformations between our platform-independent representation to the norm-aware reasoner inputs and outputs.

In our framework all agents can enter and operate in organisational contexts, which include:

1. an explicit representation of the system's organisational structure (including a model of the agents, their relationships, their goals, responsibilities and organisational norms);
2. operational descriptions of the actions in the domain;
3. a domain ontology, describing the concepts in the domain.

For the agent to be organisation-aware, it should understand and reason about the structure, work processes, and norms of the organisation.

In order to create the framework metamodel, we used the Eclipse Modelling Framework (EMF)[3]. The EMF employs XML Metadata Interchange (XMI)[4] as its default form of a model definition and can be used to generate models in UML[5] and other standards, enabling therefore interoperability with other tools and applications. In this section, with the exception of the general metamodel of Figure 3.1, which is a UML-like depiction through the EMF graphic tools[6], the metamodels that are presented have been produced by the EMF and have been transformed to XML schemas, and therefore, the notation used follows the XML Schema Definition (XSD)[7].

The complete metamodel of Figure 3.1 defines the full conceptual structure used to describe a norm-based system including the norms themselves and the agents to which they will apply in our framework. Each of the boxed frames represent a framework concept (which will be described in the remainder of this section and its subsections). Arrows indicate direct interactions between components.

[3]http://www.eclipse.org/modeling/emf/
[4]http://www.omg.org/spec/XMI/
[5]http://www.omg.org/spec/UML/
[6]We would like to make a note here, that the EMF does not provide a full equivalence to UML concepts and it has a more compact notion for Association, Aggregation and Composition relationships, using a general EReference type, depicted by the arrow with the diamond. However, it fully serves the modelling purposes of our framework and the transformations to other formats and structures intended to be used later in the thesis.
[7]http://www.w3.org/XML/Schema

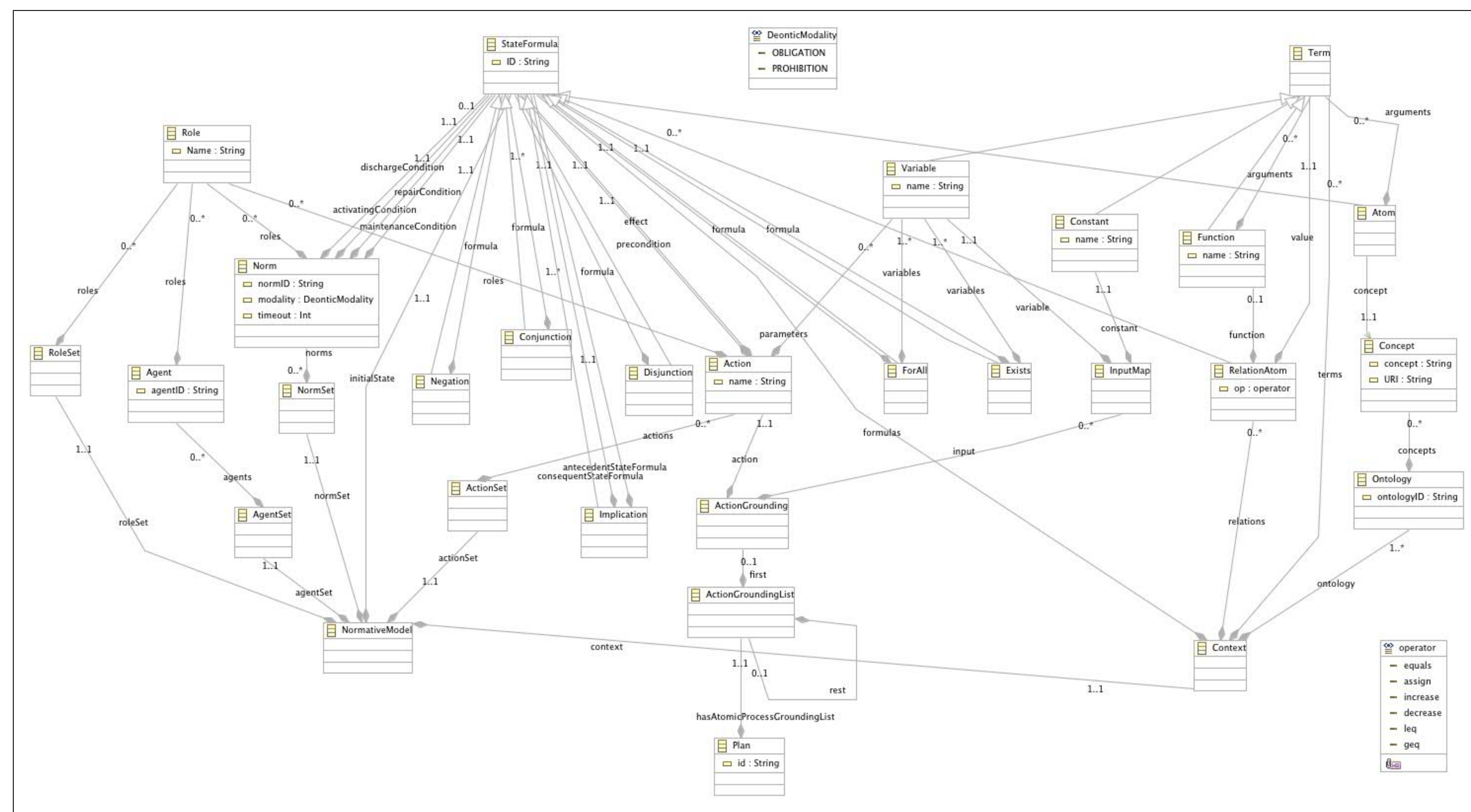


FIGURE 3.1: Overall Metamodel

Our metamodel contains two basic packages, namely **framework** (containing the basic elements such as **roleSet**, **agentSet**, **initialState**, **actionSet**, **normSet**, **plan** and **context**) and **FrameworkClasses** (containing the rest of the elements that the basic ones are broken down into).

The notation used for the diagrams is as follows (borrowed from the basic XSD notation). Each element is broken down into sub elements. The "Type" field indicates the basic type of element and which package it belongs to. The branching symbol shows how an element is broken down into sub elements. The symbol represents a sequence of elements (xsd:sequence), the symbol simpleType represents a simple type element (xsd:simpleType), the symbol complexType represents a complex type element (xsd:complexType) and the symbol attributes represents the attributes (xsd:attribute) corresponding to an element.

In order to be able to mention the elements defined in our framework later in the thesis, a **textual form** for some of them is needed. This is provided together with the analytical description of the elements.

The root of the model is the main element **framework:NormativeModel** (Figure 3.2) of the package **framework**, which contains:

- A set of *roles* (**roleSet**) of type **framework:RoleSet**
- A set of *agents* (**agentSet**) of type **framework:AgentSet**
- The current *state of affairs* (**initialState**) of type **FrameworkClasses:StateFormula**
- A set of (domain) *actions* (**actionSet**) of type **framework:ActionSet**
- A set of *norms* (**normSet**) of type **framework:NormSet**
- An (organisational) *context* (**context**) of type **framework:Context**, containing information on the domain to be modelled

We will describe each of them in the following sections.

3.2.1 Context

The **framework:Context** type (see Figure 3.3) contains the environment's extra knowledge needed for the agents' reasoning. It contains one or multiple **ontologies** of type **FrameworkClasses:Ontology**, a set of zero or more **terms** of type **FrameworkClasses:Term**, a set of zero or more **formulas** of type **FrameworkClasses:StateFormula** and a set of zero or more **relations** of type **FrameworkClasses:RelationAtom**. These terms (variables, etc.) combined with the ontology concepts can used to make formulas (atoms, conjunctions, etc.) describing the state of the agent.

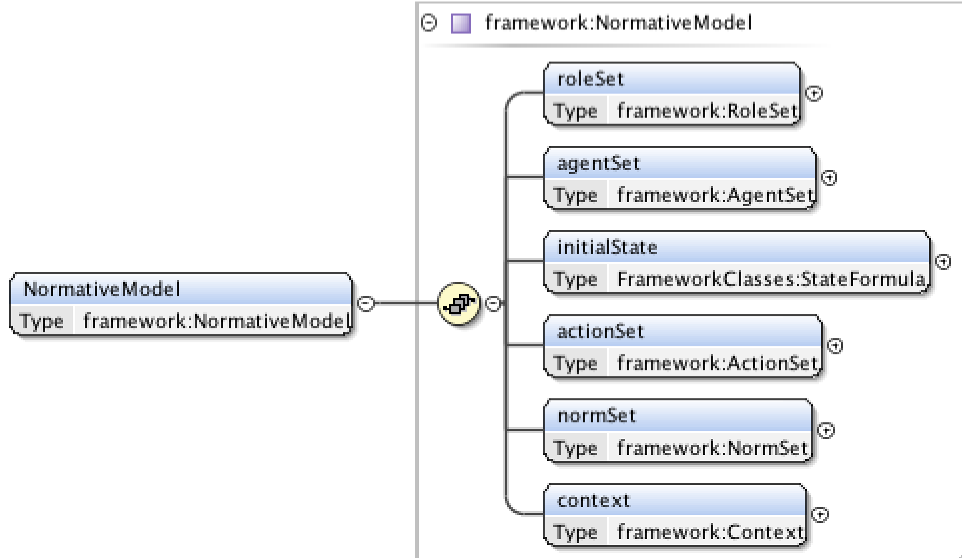


FIGURE 3.2: Normative Metamodel

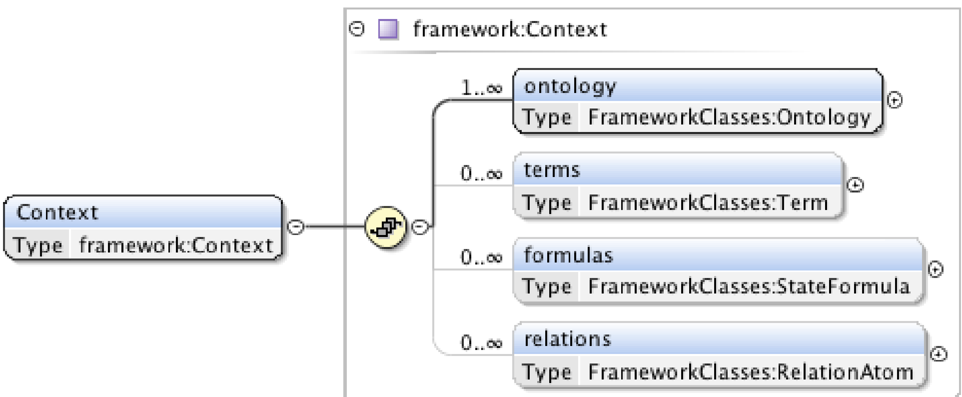


FIGURE 3.3: Context

3.2.1.1 Ontology and Concept

The **FrameworkClasses:Ontology** (or simply **ontology**) type (Figure 3.4) predefines all the concepts that are used by the framework. The purpose of this is for generic, non-application-dependent information to exist within the framework and be adopted by the agents. An ontology consists of a unique **ontologyID** and zero or more **concepts** of type **FrameworkClasses:Concept**.

Concepts are the main components of every ontology. In our framework, the type **FrameworkClasses:Concept** is defined in a simple way (Figure 3.5), having two attributes, **concept** and **URI**. The first is a unique naming of every concept in order to make it identifiable, while the second (optional) is a Uniform Resource Identifier where a complex concept definition could exist and be available on the net. The textual form of a concept is simply its name. For example, a concept might be: `driving`.

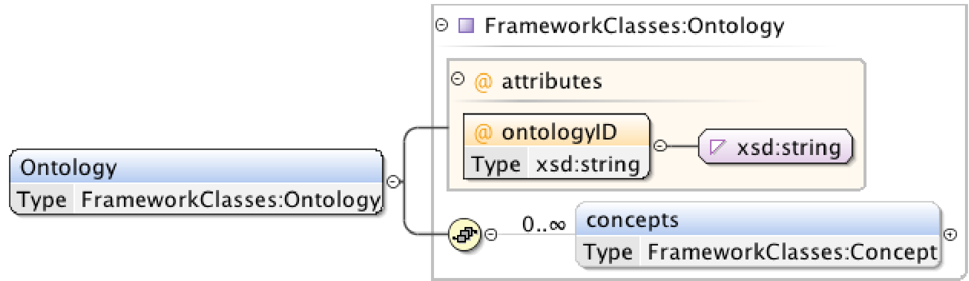


FIGURE 3.4: Ontology

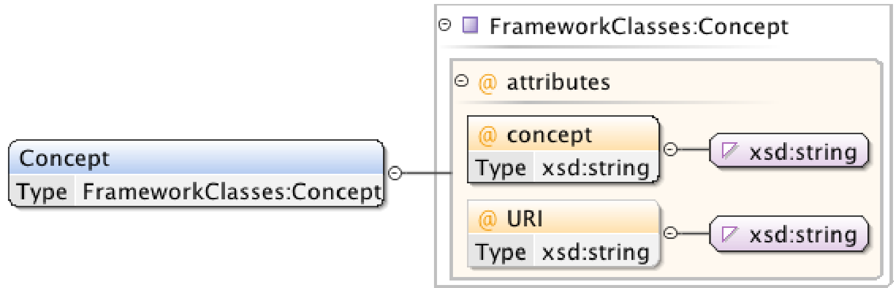


FIGURE 3.5: Concept

3.2.1.2 Terms

Terms (Figure 3.6) are simple elements used as arguments when defining a **FrameworkClasses:Atom** (atoms are the elements that are used to form the state of the world, as will be explained in Section 3.2.1.3) or when defining a **FrameworkClasses: Function**. The type **FrameworkClasses:Term** is defined as an abstract type and can be implemented as a **FrameworkClasses:Variable** (Figure 3.7), **FrameworkClasses: Constant** (Figure 3.8) or **FrameworkClasses:Function** (Figure 3.9).

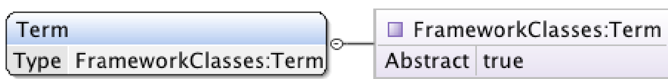


FIGURE 3.6: Term

The type **FrameworkClasses:Variable** (or simply **variable**) is defined simply by a unique **name**. It might be used as an argument in an **FrameworkClasses:Function** or a **FrameworkClasses:Atom**. The textual representation of a variable is its **name**[8]. For example, a variable can be: `street1`.

The type **FrameworkClasses:Constant** (or simply **constant**) too is defined simply by a unique **name**. It might be used in cases a constant (for example a number) is needed as an argument in a **FrameworkClasses:Function** or a **FrameworkClasses:Atom**. The textual representation of a constant is simply its **name**. For example, a constant can be: `maria`.

[8] As we will see in Section 4.5, a "'?" might precede a variable in order to make it more distinguishable.

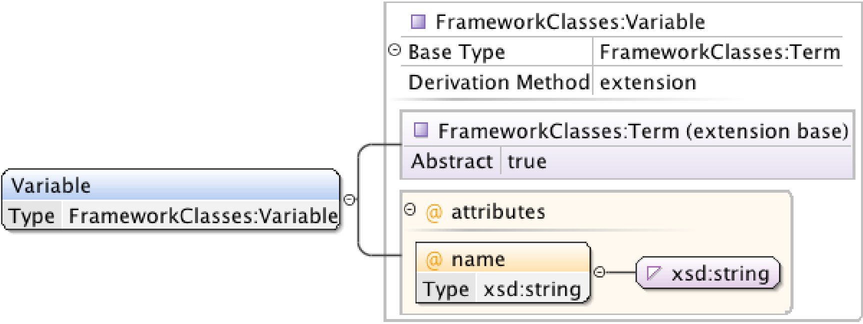


FIGURE 3.7: Variable

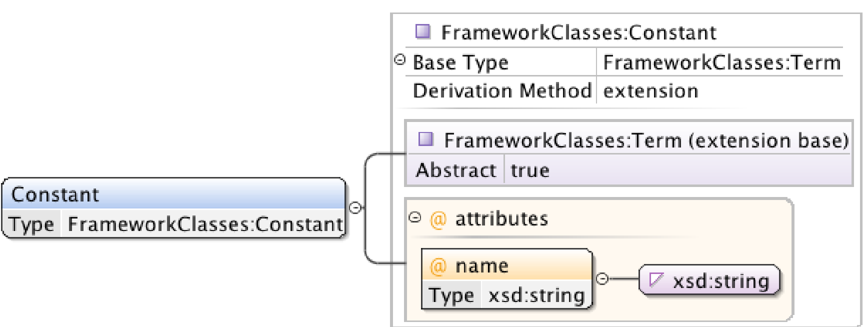


FIGURE 3.8: Constant

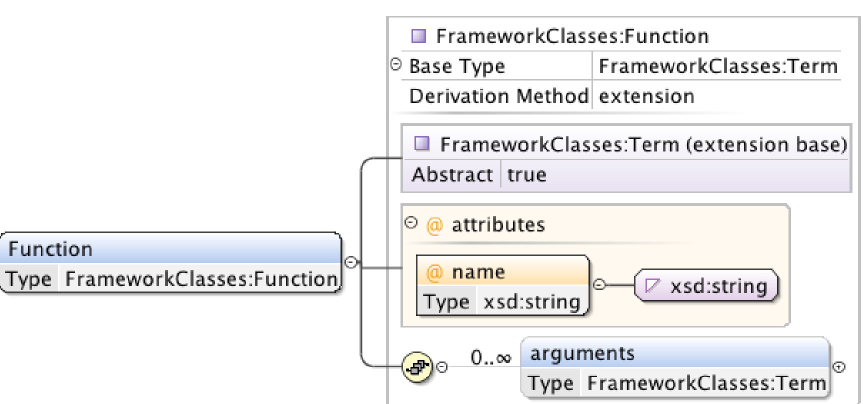


FIGURE 3.9: Function

The type **FrameworkClasses:Function** (or simply **function**) has a unique **name** and zero or more **arguments** of type **FrameworkClasses:Term**. The textual representation of a function is the **name** followed by the **arguments** in a parenthesis, separated by commas. For example, a function can be: `speedLimit(street1)`.

3.2.1.3 Formulas, Relation Atoms and State of Affairs (StateFormula)

During the enactment/execution of the multi-agent system, there must be an account of the current state of affairs. In principle, the state of affairs consists of the combination of the fluents/atoms/predicates holding and the effects of all actions executed

during a shared plan/workflow[9].

A **State of Affairs** is *bootstrapped* with some state formulas which hold at some point in time. Since states of affairs should be able to model parts of the world state instead of the full world state and agents usually do not receive the full state at the system they operate in, states of affairs can be consequently seen as **Partial State Descriptions (PSDs)**. PSDs, as the name suggests, describe partial aspects of the world which hold true at a certain point in time; in other words, it is a representation of (parts of) the full state of affairs in terms of the properties of the state (e.g. goods are sold, thermostat is on, room is cold, etc.) that hold at a certain point in time. A PSD is represented as a logical formula. PSDs will be used to describe the norm conditions and to give meaning to plan enactments. Agents use PSDs as a set of beliefs, that is, what is (believed to be) true at a point in time. Actions (described in Section 3.2.5) change the state of affairs. States of affairs relate to each other via actions: an action performed on a PSD gives rise to another PSD.

Partial State Descriptions are defined as follows:

- A (propositional) Atom is a Partial State Description
- A Relation Atom[10] is a Partial State Description
- If φ and ψ are Partial State Descriptions, then so are $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$ and $\varphi \rightarrow \psi$
- If ϕ is a Partial State Description and $x_1, x_2, \ldots, x_n$ are variables, then $\forall x_1, x_2, \ldots, x_n : \phi$ and $\exists x_1, x_2, \ldots, x_n : \phi$ are Partial State Descriptions

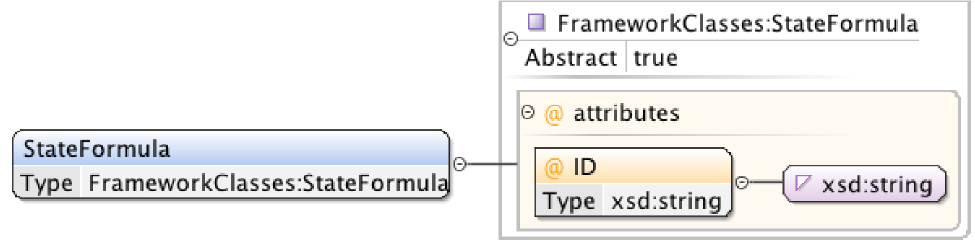


FIGURE 3.10: StateFormula

Reflecting the above definition, in our framework states of affairs are defined through a **FrameworkClasses:StateFormula** (or simply **StateFormula**, see Figure 3.10) as an

[9]Clearly, a global state of affairs shared by the agents in the (organisational) environment is required for the appropriate enactment of the generic framework, as the effects of an action executed by an agent may be required as a precondition of an action by another agent. As our framework is platform-independent and technology-independent, it does not impose a mechanism for the agents to share the global state of affairs. From now on we will suppose the existence of a simple distributed protocol whereby each agent keeps its own version of the current state of affairs and propagates relevant changes in its beliefs on the environmental state to other agents. When an agent carries out one action, it updates its version of the state of affairs and sends a broadcast to all/some other agents with the updates they should carry out. This will ensure the agents' versions of the current state of affairs are all eventually consistent. It is important to note that our framework is independent from the actual state-of-affairs sharing mechanism used, and therefore agent designers can change this simple protocol into a more advanced mechanism without affecting our framework.

[10]We will explain this later in this section.

abstract element which might have several subtypes: **atom**, **relation atom**, **negation** of atoms and **conjunction**, **disjunction**, **implication**, **forall** and **exists** connecting atoms or StateFormulas.

The **FrameworkClasses:Atom** (or simply **atom**, see Figure 3.11) type is defined by a unique **ID**, a **concept** of type **FrameworkClasses:Concept** and a number (possibly zero) of **arguments** of type **FrameworkClasses:Term**. The textual form to express an atom is by its **concept**, followed by the **arguments** in parenthesis. An example of an atom can be: `driving(maria, vehicle)`.

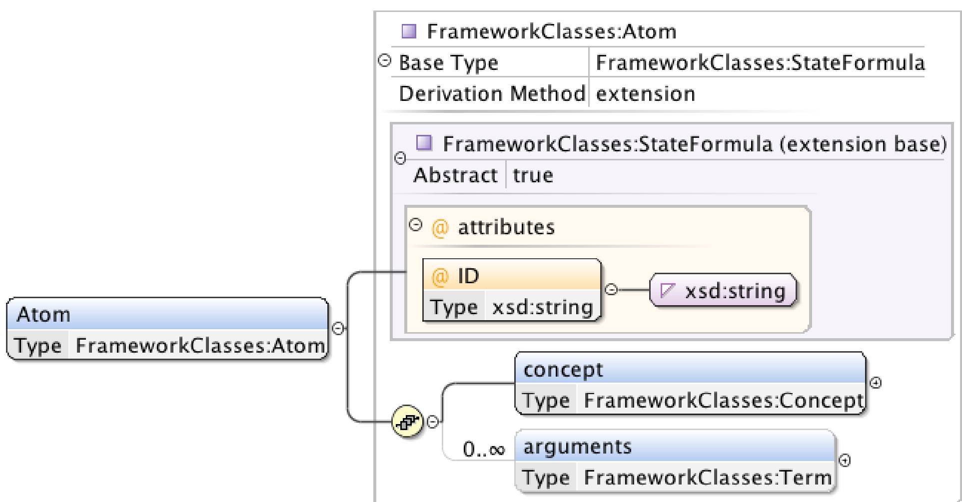


FIGURE 3.11: Atom

The **FrameworkClasses:RelationAtom** (see Figure 3.12) type is defined by a unique **ID**, an **op**erator of type **FrameworkClasses:operator**, a **function** of type **Framework-Classes:Function** and a **value** of type **FrameworkClasses:Term**. A relation atom represents special relationships that tie a function to a value. There are 6 types of operators: **assign** (or alternatively :=), **increase** (or alternatively or +=), **decrease** (or alternatively or -=), **equals** (or alternatively =), **greater or equal** (geq, or alternatively or ¿=), **less or equal** (leq, or alternatively or ¡=). The textual form to express a relation atom depends on its type. For assignment, we write assign followed by the function and then the value. For the rest, we write the function first, then the name of the operator and then the value. An example of such an atom can be: `assign money(jack) 40`, another can be `money(jack)+=100` and another can be `money(jack) geq money(maria)` .

The **FrameworkClasses:Negation** (or simply **negation**, see Figure 3.13) type is defined by a unique **ID** and a formula of type **FrameworkClasses:StateFormula**. The textual form to express a negation is by the use of the symbol ¬ followed by the textual form of the **formula** itself. An example of a negation might be: `¬driving(maria, vehicle)`.

The **FrameworkClasses:Conjunction** (or simply **conjunction**, see Figure 3.14) type is defined by a unique **ID** and one or more **formulas** of type **FrameworkClasses:**

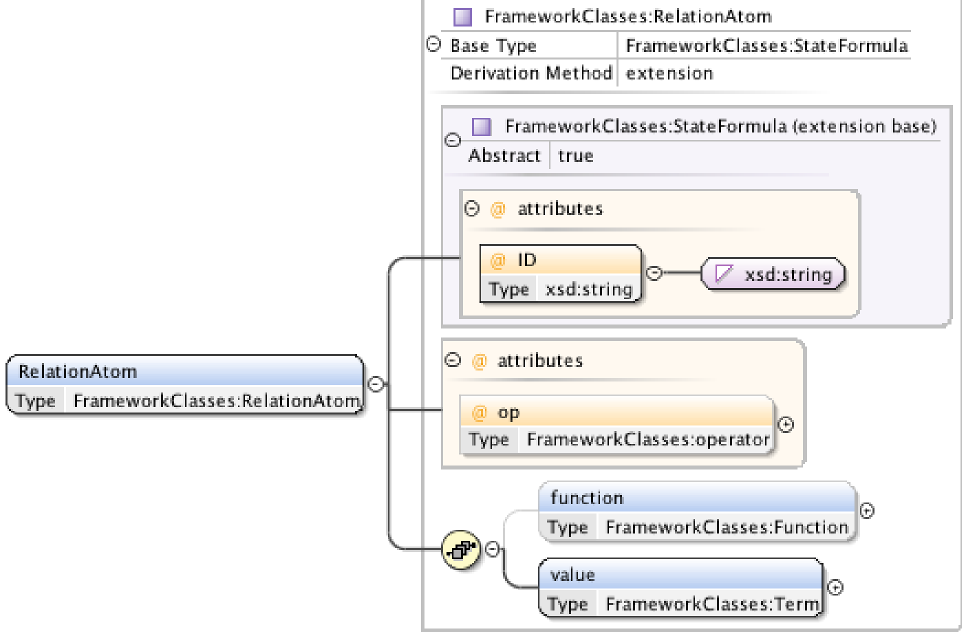


FIGURE 3.12: Atom

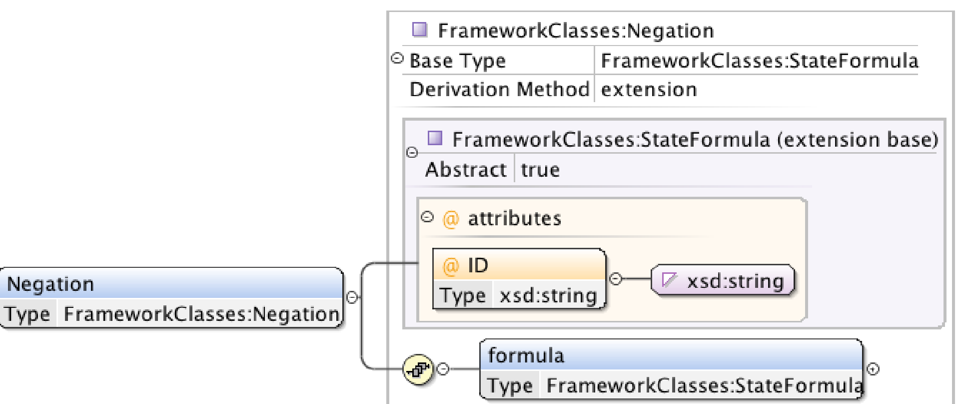


FIGURE 3.13: Negation

StateFormula. The textual form to express a conjunction is by the use of the symbol `and` (or sometimes ∧) connecting the textual form of each of the **formula**s. An example of a conjunction might be: `¬driving(maria, vehicle) and fasterThan (vehicleSpeed(vehicle),80)`.

The **FrameworkClasses:Disjunction** (or simply **disjunction**, see Figure 3.15) type is defined by a unique **ID** and one or more formulas of type **FrameworkClasses:State Formula**. The textual form to express a disjunction is by the use of the symbol `or` (or sometimes ∨) connecting the textual form of each of the **formula**s. An example of a disjunction might be: `¬driving(maria, vehicle) or fasterThan( vehicleSpeed(vehicle),80)`.

The **FrameworkClasses:Implication** (or simply **implication**, see Figure 3.16) type is

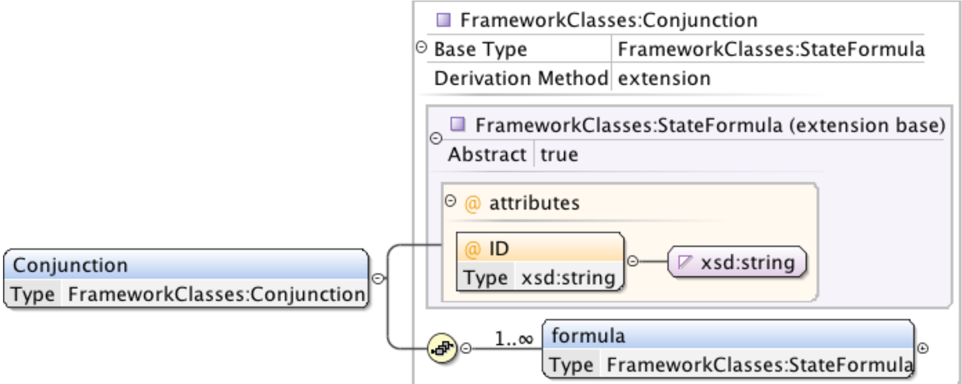


FIGURE 3.14: Conjunction

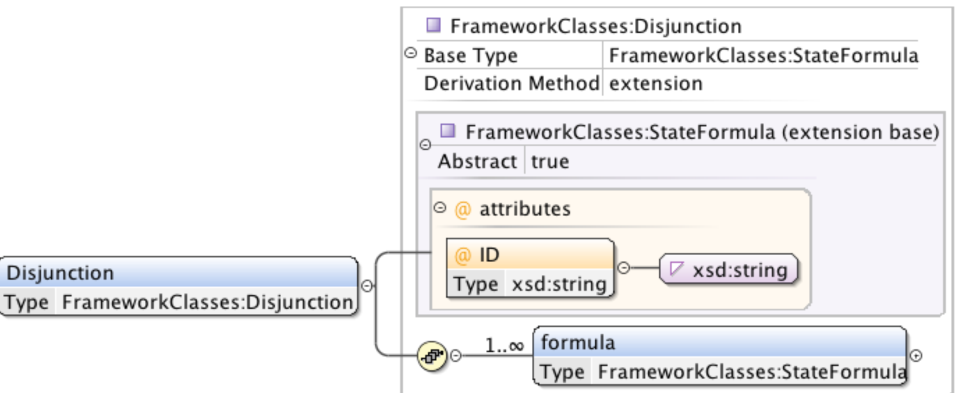


FIGURE 3.15: Disjunction

defined by a unique **ID** and two formulas of type **FrameworkClasses:StateFormula**, namely **antecedentStateFormula** and **consequentStateFormula**. The textual form to express an implication is by the use of the symbol $\rightarrow$ connecting the textual form of the **antecedentStateFormula** with the **consequentStateFormula**. An example of an implication might be: ¬`driving(maria, vehicle)` $\rightarrow$ ¬ `at_home(maria)`.

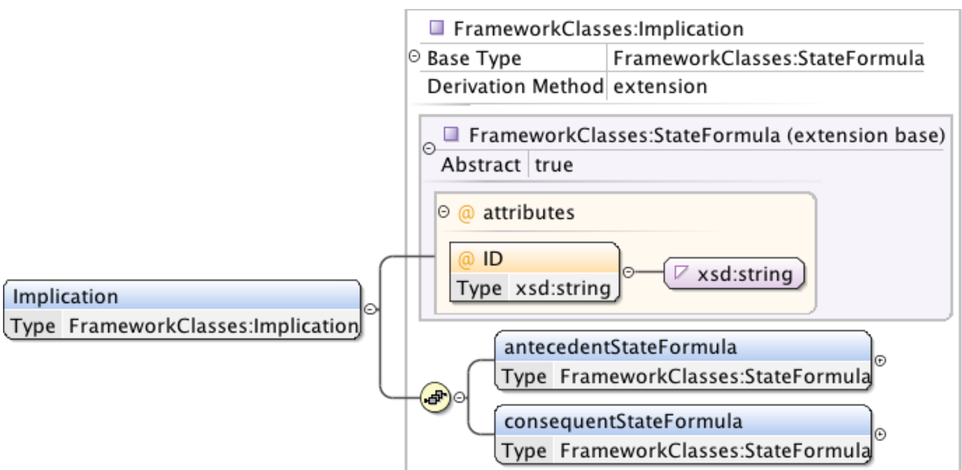


FIGURE 3.16: Implication

The **FrameworkClasses:ForAll** (or simply **forall**, see Figure 3.17) type represents the universal quantifier. It is defined by a unique **ID**, one or more **variables** of type **FrameworkClasses:Variable** and a **formula** of type **FrameworkClasses:StateFormula**. The textual form to express this quantifier is by the use of the symbol ∀ followed by the **variables**, a ":" and then the textual form of the **formula**. An example of a forall might be: ∀`p: driving(p, vehicle).`

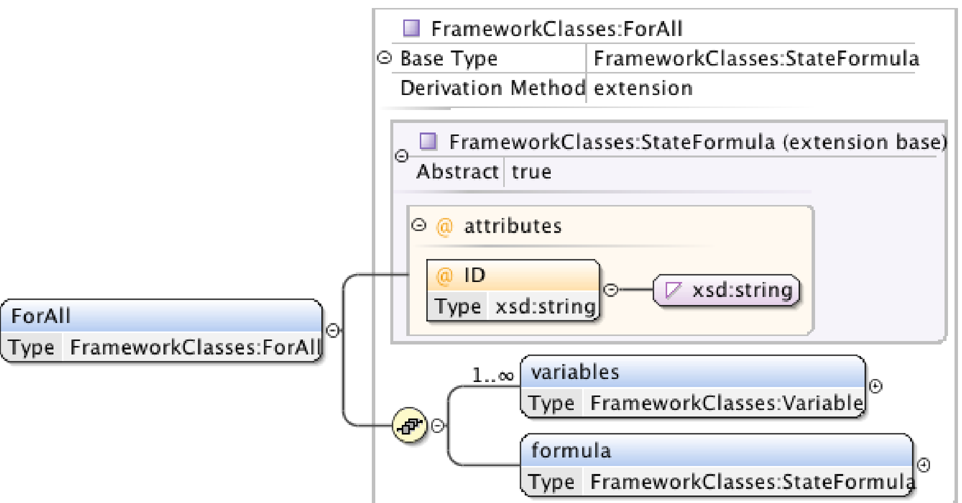


FIGURE 3.17: Universal Quantification

The **FrameworkClasses:Exists** (or simply **exists**, see Figure 3.18) type represents the existential quantifier. It is defined by a unique **ID**, one or more **variables** of type **FrameworkClasses:Variable** and a **formula** of type **FrameworkClasses:StateFormula**. The textual form to express this quantifier is by the use of the symbol ∃ followed by the **variables**, a ":" and then the textual form of the **formula**. An example of an exists might be: ∃`p1,p2: ¬driving(p1, vehicle) and ¬ at_home(p2).`

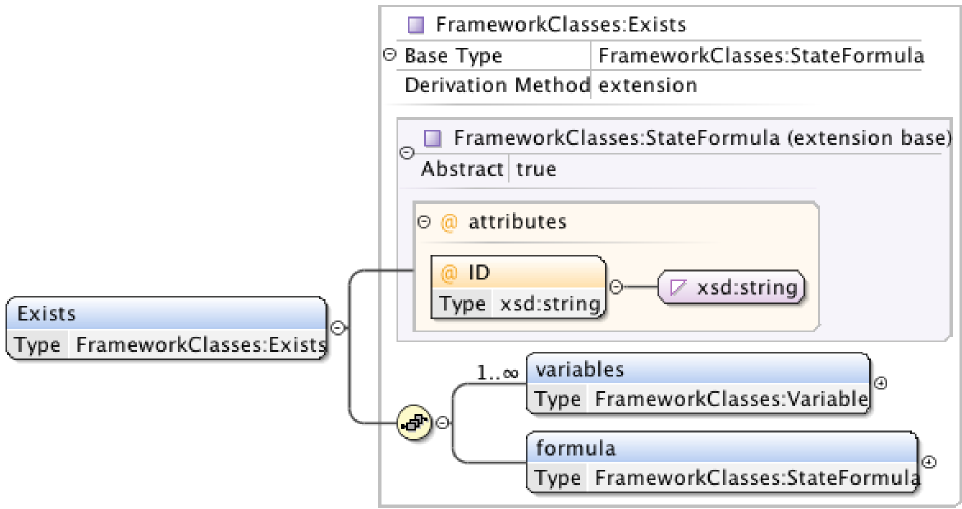


FIGURE 3.18: Existential Quantification

3.2.2 Roles

The **FrameworkClasses:Role** (or simply **role**, see Figure 3.19) element is used to assign roles to the agents. A role is a constant. Each role consists of a unique **Name** that identifies it. As we will see in Section 3.2.3, agents are associated with one or more roles. We say that for each one of the roles associated with an agent, the agent *enacts* that role. Roles are what bind agents to norms. As we will see in Section 3.2.6 each norm addresses specific roles, and only the agents enacting those roles are bound to that norm.

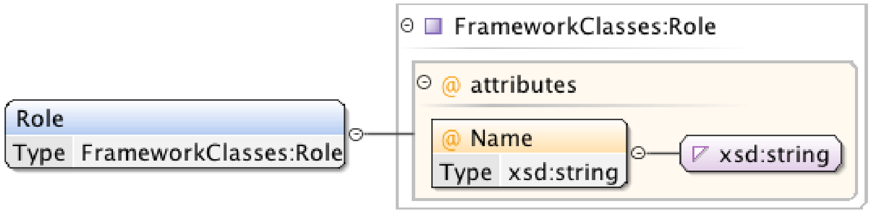


FIGURE 3.19: Roles

The **framework:RoleSet** (see Figure 3.20) element consists of a set **roles** of **FrameworkClasses:Role** type elements.

FIGURE 3.20: Role Set

3.2.3 Agents

A **FrameworkClasses:Agent** (or simply **agent**, see Figure 3.21) is an entity of the organisation representing an individual. It is ascribed the properties of being autonomous, proactive, flexible (decision-making) and social. We envision the framework supporting many different agent types, and assume only that agents are able to send and receive messages[11]. Since agents may have norms that apply to them, we must be able to uniquely identify agents. We therefore assume that an agent has some unique **agentID** that is a constant and a set of assigned **roles** of type **FrameworkClasses:Role** that it enacts. We can assume that the relation between an agent and a role can be described as $enacts(\alpha, r)$ whenever the set of roles of agent α contains role r[12].

[11]In the case of service-oriented applications our framework considers each service as an agent with, at least, the following capabilities: reactiveness, social ability and (limited or none) proactivity.

[12]Such a relation in most frameworks can be translated to a predicate *enacts*. We will use this later in Chapters 4 and 5 to indicate the established relationship between an agent and a role.

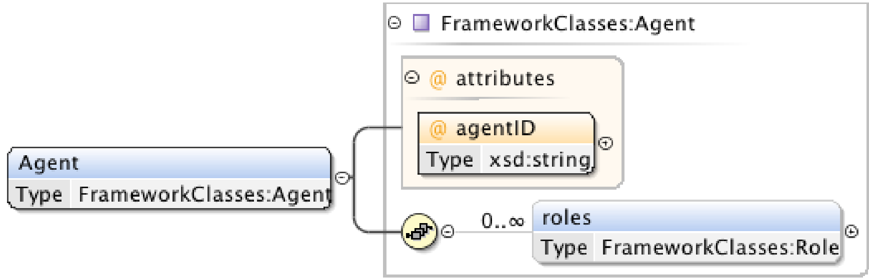


FIGURE 3.21: Agents

The **AgentSet** (see Figure 3.22) element consists of a set **agents** of **FrameworkClasses: Agent** type elements.

FIGURE 3.22: Agent Set

3.2.4 Initial State

The **initialState** depicted in Figure 3.2 is used to start the agent's normative reasoning procedure and consists of the fluents/atoms/predicates holding at the beginning of an execution. This will be represented, like all other states, by a **FrameworkClasses:State Formula** (see Section 3.2.1.3).

3.2.5 Actions

A **FrameworkClasses:Action** (or simply **action**, see Figure 3.23) corresponds to some capability of an agent. Each action has a unique **actionName** in order to be identified. A set of **parameters** of type **FrameworkClasses:Variable** is also associated to each action. Additionally, each action can be associated with a number of **roles** of type **FrameworkClasses:Role**, specifying what the agents that enact those, are able to perform. Agents that do not enact one role of the **roles** associated with the action do not have access to it.

Actions are defined through logic formulas expressing *preconditions* and *effects* which hold before and after the action is executed[13]. Actions are an essential element in our framework as they lead to the formation of execution paths to achieve the goals of an agent. The basis of action representation is PSDs, and subsequently state formulas,

[13]We borrow this standard representation of actions used by the Classical Planning community.

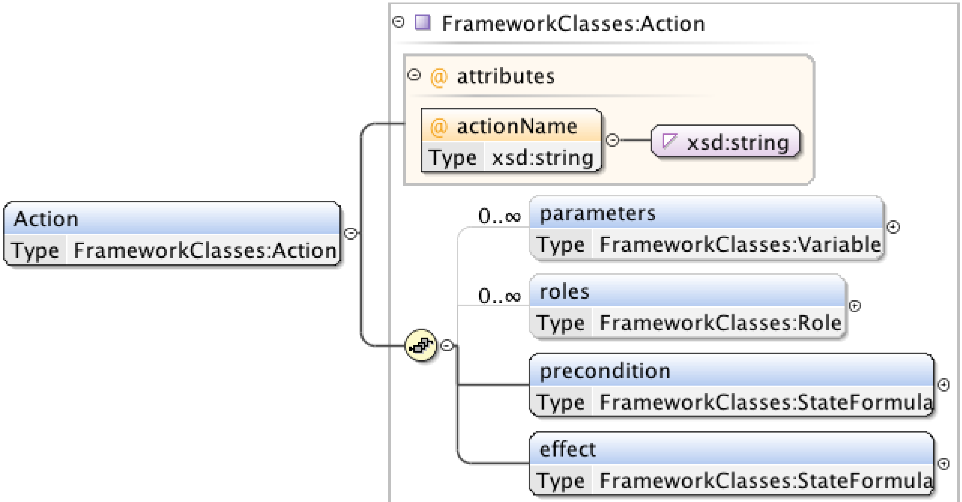


FIGURE 3.23: Actions

which are used to represent both the precondition and the effects of the action. Therefore, the **precondition** of an action specifies the state of affairs (of type **FrameworkClasses:StateFormula**) in which the action can be performed. In a similar way, the **effect** of an action specifies the state of affairs (of type **FrameworkClasses:StateFormula**) to which the execution of the action leads.

The textual representation of an action will be its **actionName** followed by the **parameters**, and then the textual representation of the **precondition** and the **effect**. An example of an action might be:

```
ParkCar
parameters:    p v
precondition:  insideCar(p,v) and nextToEmptySpace(v)
effect:        carParked(v)
```

The **ActionSet** (see Figure 3.24) element consists of a set of **actions**, of type **FrameworkClasses:Action** elements.

FIGURE 3.24: Action Set

3.2.6 Norms

FrameworkClasses:Norm (or simply **norm**, see Figure 3.25) expresses deontic restrictions coming from an organisational specification and provide guidelines over the agents' behaviour. These deontic statements such as obligations and prohibitions as

well as properties related to the norm's lifecycle are formalised and implemented through a deontic framework[14].

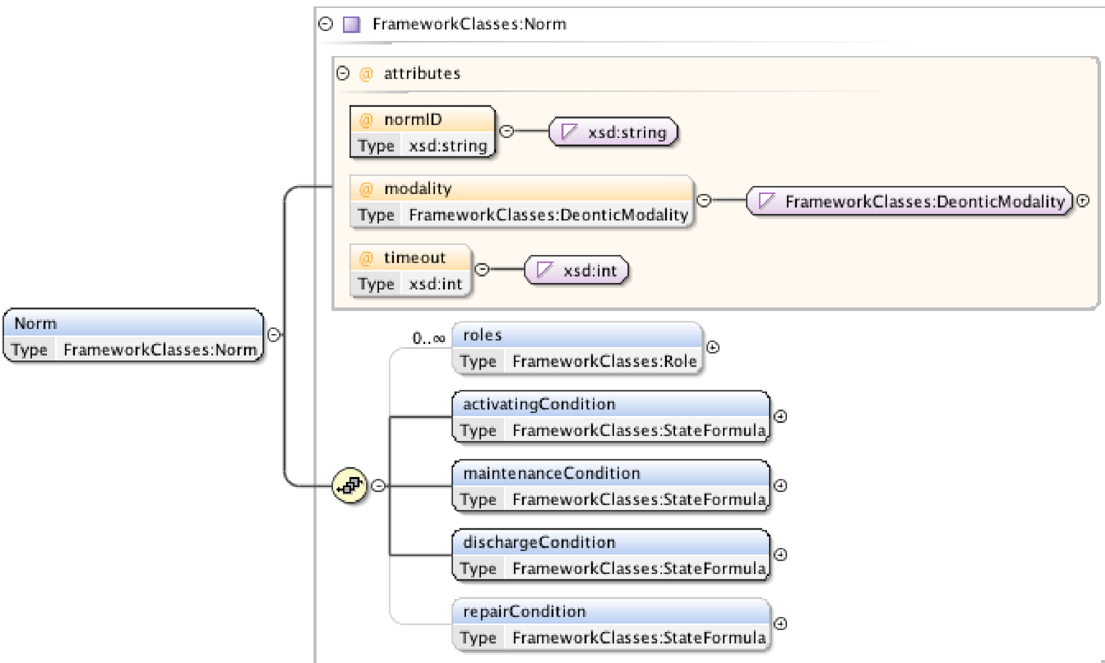


FIGURE 3.25: Norms

The **framework:NormSet** (see Figure 3.26) element consists of a set **norms** of **FrameworkClasses:Norm** type elements.

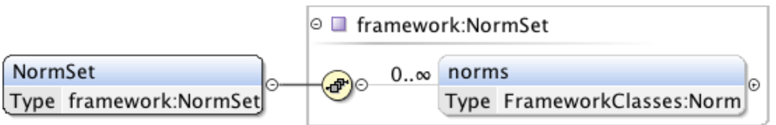


FIGURE 3.26: Norms Set

We now provide details on how norms are specified. Norms are defined through a set of logic formulas expressing what is obliged or prohibited for certain roles/agents in the organisation. Instead of using a deontic representation based on modal logic, our representation focuses on explicitly representing the operational aspects of norms (their activation, discharge, fulfilment or violation) in first-order logic in order to ease its use at runtime. As in actions, the basis of our norm representation is the **FrameworkClasses:StateFormula**.

In addition to the previous, a violation penalty proposition is introduced as a **FrameworkClasses:StateFormula** to indicate the satisfaction of the penalty of the norm in the case of a violation of a norm.

As a consequence, a norm is then composed of the following elements:

[14]In Section 4.4 we show how our norm representation connects to Standard Deontic Logic and Dyadic Logic.

- **normID** (or $\boldsymbol{n}$): a unique id to identify the norm
- **roles**: a set of roles of type **FrameworkClasses:Role** to which the norm applies
- **modality**: the type of norm (O for obligation, F for prohibition, P for permission)
- **activatingCondition** [15] (or, for short, $\boldsymbol{f_n^A}$): a condition of type **FrameworkClasses:StateFormula**
- **maintenanceCondition** [15] (or, for short, $\boldsymbol{f_n^M}$): a condition of type **FrameworkClasses:StateFormula**
- **dischargeCondition** [15] (or, for short, $\boldsymbol{f_n^D}$): a condition of type **FrameworkClasses:StateFormula**
- **repairCondition** [15] (or, for short, $\boldsymbol{f_n^R}$): a condition of type **FrameworkClasses:StateFormula**
- An optional field $\boldsymbol{timeout_n}$, representing the time limit within which action needs to be taken to have a norm repaired in the case of a violation

Let us look at these elements in more detail taking into account a simple norm example: "In case of evacuation order, the Police are obliged to evacuate the site" (formalised as $eorder \rightarrow O_p evac$). The modality of the deontic statement expresses the deontic 'flavour' of the norm (and thereby establishes whether an agent should attempt to fulfil the norm or, on the contrary, avoid the prohibition). The modality in the example is an obligation (O), but could also have been a prohibition (F) or permission (P). The roles set ties the norm to the agent role (or roles) that should adhere to the norm. In the case of the example the roles set contains *police* to denote that the norm applies to agents enacting that role.

The textual form for the representation of the norms is the **normID**, followed by the **modality**, the list of **roles** and then the textual form of the **activatingCondition**, **maintenanceCondition**, **dischargeCondition** and **repairCondition**. An example of the above elements of a norm where a driver in the city is supposed to stay out of the bus lanes (expressed informally as the language has not been yet fully defined) would be:

```
normID: StayOutOfBusLane
modality: O
roles: driver
activatingCondition:  inTheCity(p) and driving(p, v)
maintenanceCondition: ¬onBusLane(v)
dischargeCondition:   ¬inTheCity(p) or ¬driving(p, v)
repairCondition:      finePaid(p,fine)
```

Obligations

[15]This element is explained later in this section, on page 89.

Obligations are probably the most commonly used type of norm. An agent having an obligation should ensure that the obligation's maintenance condition holds. If this does not occur, the obligation is violated.

Prohibitions

Amongst different views, prohibitions may be seen as obligations on preventing a certain state of affairs from coming about. We follow the Standard Deontic Logic view ($Fx = O\neg x$) [von Wright, 1951, 1971], and treat prohibitions as a form of obligation. Within our framework, a prohibition is represented as an obligation in which the maintenance condition is negated. For example, a prohibition of the form "the agent is prohibited from entering a building on fire" may be transformed into an obligation "the agent is obliged not to enter a building on fire".

Permissions

Researchers have dealt with permissions in a number of ways [Kollingbaum, 2005]. Standard Deontic Logic suggests the interpretation of permissions as exceptions to obligations (permission seen as a dual of obligation). That is, if an obligation requires A to occur, and a permission allows B to occur instead, then if B occurs, no violation of the obligation occurs. Another way of viewing permissions is by assuming that anything that is not explicitly permitted within the system is prohibited (permission seen as absence of prohibition) [von Wright, 1951; Hilpinen, 1971]. This case could be represented, informally, as "An agent is obliged to undertake no actions that affect the world". Permissions then form exceptions to this obligation, allowing this interpretation of permissions to be reduced to the first approach.

In our framework, we discard permissions and use only obligations and prohibitions. We explicitly assume that everything is permitted unless otherwise specified. This choice is based on the wide known legal principle "*Nulla poena sine lege (No penalty without a law)*" stating that one cannot be punished for doing something that is not imposed by law. Written by Paul Johann Anselm Ritter von Feuerbach as part of the Bavarian Criminal Code in 1813, this principle has been accepted and upheld by the penal codes of constitutional states, including virtually all modern democracies.

Additionally, we feel that permissions are not essential in our model, but only complementary elements that can be expressed by the obligations and prohibitions. Possible complications when dealing with their semantics as well as their limited use in the law intuitively bring us to the conclusion that a 'free world', allowing all actions to take place under the appropriate conditions, is a sufficient set for our normative agents.

Activating, maintenance, discharge and repair conditions

The **activatingCondition** (activating condition), **maintenanceCondition** (maintenance condition), **dischargeCondition** (discharge condition) and **repairCondition** (repair condition) are specified as **FrameworkClasses:StateFormula**s, denoting the conditions that express when the norm gets activated, violated, discharged and repaired. They

add operational information to the norm, to simplify its verification and enforcement. They work as follows: the activating condition specifies when a norm becomes active (*eorder* in the example), specifying the state of affairs in which the norm is triggered (and must henceforth be checked for completion/violation). The discharge condition specifies when the norm has been discharged[16] (in the example given, the discharge of the norm would be specified as *evac*). The maintenance condition is needed for checking violations of the norm; it expresses the state of affairs that should hold all the time between the activation and the discharge of the norm (in the example the maintenance condition is *evac*). Finally, the repair condition is used to indicate a desired repair state to be reached in case of a violation of the norm. In case of a violation, a timeout can be used to indicate the time limit within which the norm should reach the repair state.

The lifecycle and dynamics of norms will be further detailed in Section 3.4 and be given full operational semantics later in Chapters 4 and 5. In essence, when a norm has been activated, has not yet been discharged and the maintenance condition is not fulfilled, a violation of the norm happens. Such a violation, typically represented as a special proposition (e.g. $violated(StayOutOfBusLane)$) or by a flag being raised, represents that the system is in a state where the norm was not adhered to, and some actions have to be undertaken (e.g. to punish the agent breaking the norm, or to repair the system). Whenever a norm is violated, the norm continues being active but in addition, the agent has to fulfil the state described by the repair condition.

3.2.7 Plans

A **framework:Plan** (or simply **plan**) consists of lists of actions (indicating in this way an execution path), performed sequentially by some agent. In order to specify a plan, each action belonging to it must have its parameters appropriately grounded. Formally (see Figure 3.27), a plan consists of a unique **id**, used for identification purposes, and an element **hasAtomicProcessGroundingList** of type **FrameworkClassess:Action GroundingList**.

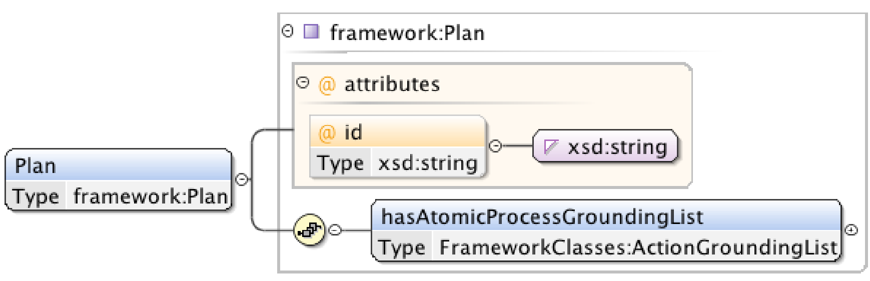


FIGURE 3.27: Plan

[16]In some cases, norms can become "obsolete" even though they are not fulfilled. At this point, the discharge of a norm can be considered synonymous to the fulfilment of the norm. In Chapter 4 the fulfilment of a norm will be further discussed and the relationship to the norm's particular instances will be shown.

The type **FrameworkClassess:ActionGroundingList** (see Figure 3.28) is a list of action groundings. It is defined by the **first** element, of type **FrameworkClasses:Action Grounding** and the **rest**, containing the rest of the list of action groundings.

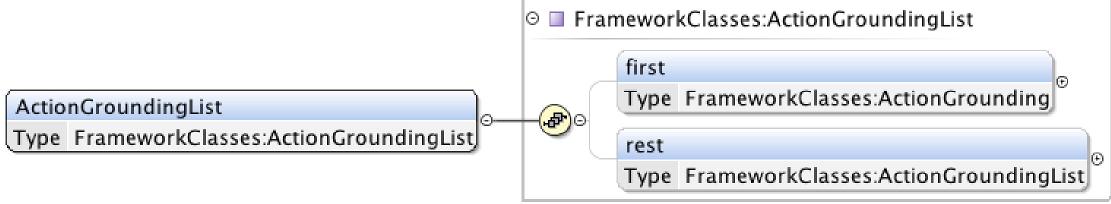


FIGURE 3.28: Action Grounding List

An **action grounding**, represented by the type **FrameworkClasses:ActionGrounding** (see Figure 3.29) represents an **action grounding** and grounds each of the parameters of an action. It consists of an **action** element of type **FrameworkClasses:Action** (the action to be grounded) and zero or more **input**s of type **FrameworkClasses:Input Map**. The action grounding should contain one input element for each parameter of the specific action.

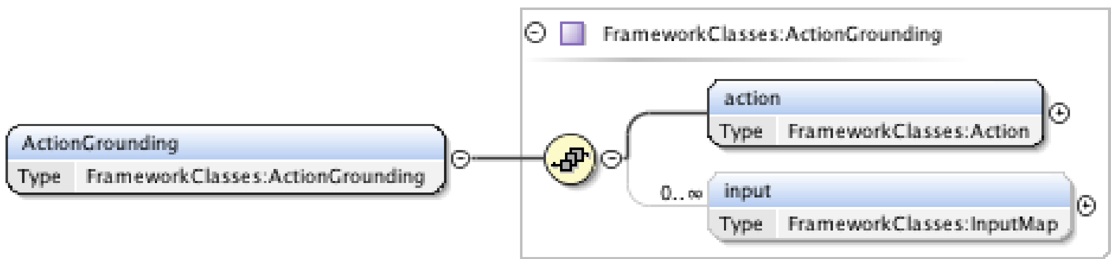


FIGURE 3.29: Action Grounding

The **FrameworkClasses:InputMap** element (see Figure 3.30) represents an **input map** and matches exactly one **variable** of type **FrameworkClasses:Variable** with exactly one **constant** of type **FrameworkClasses:Constant**.

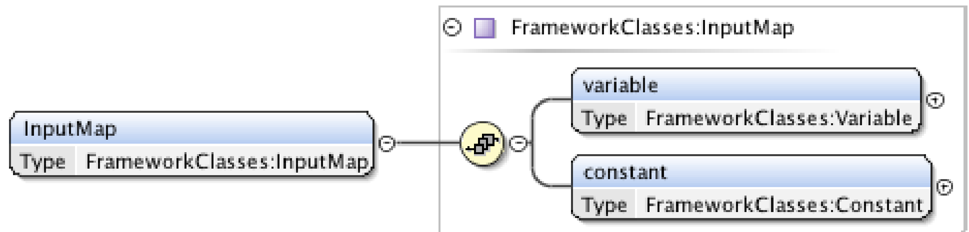


FIGURE 3.30: Input Map

The textual representation of the **FrameworkClasses:InputMap** is the **variable** followed by the substitution symbol ← and then the **constant**. For example an input map can be: `p←maria`.

The textual representation for the **FrameworkClasses:ActionGrounding** is the name of the **action** followed by the textual form of each **input** in parenthesis. An example

of an action grounding would be: `driveCar(p←maria,v←bmw)`. A shorter version for a textual representation of the action grounding is the one where the variables are omitted. Then, the same example would become: `driveCar(maria,v)`.

The **FrameworkClasses:ActionGroundingList** is textually represented as the list of the textual representations of each action grounding. An example of such an action grounding list could be: `driveCar(maria,v),parkCar(maria,v)`.

Finally, the **framework:Plan** textual representation will be its **id**, followed by ':' and then the textual representation of its **hasAtomicProcessGroundingList** element. In this way, an example of a plan would be: `plan1:driveCar(maria,v), parkCar (maria,v)`.

3.3 Architecture

Our framework architecture can be seen in Figure 3.31. All agents operate within an organisational context. The *Organisation* provides a social context for agents, specifying the organisational conventions that affect interaction (ontology terms, action descriptions and norms) and including mechanisms to allow the structural adaptation of systems over time. The *World State* models the description of the world, as seen from the organisational perspective.

3.3.1 Components

The architecture consists of several interconnected components as seen in Figure 3.31. It includes two metamodels, the norms metamodel and the actions metamodel which respectively abstract the norms and actions. Ontology terms need no special metamodel as we use OWL [Antoniou and van Harmelen, 2003] as a basis. Metamodel instantiations of norms and actions (of a domain) defined as metamodel elements of Section 3.2 are called models and can be visualised by using tools.

Actual source code, which will serve as input to the reasoner, is created automatically by applying predefined transformations from the source model with respect to the metamodels. The code (for norms and actions) is produced in the necessary format in order to be processed by the reasoner. Our framework elements (such as Action, Function, Atom, Variable, Constant, StateFormula, Plan) of Section 3.2 can be directly translatable from and to the syntax of most planning mechanisms[17].

A BDI agent stands as the basis of the architecture[18]. It is organisation-aware, that is, it is aware of the relevant system objectives, roles and norms. For the agent to be organisation-aware, it is necessary for it to be able understand and reason about the

[17] In the case of our thesis TLPLAN and PDDL, as we will see in Chapters 4 and 5 respectively

[18] It will be described in Section 3.3.2.

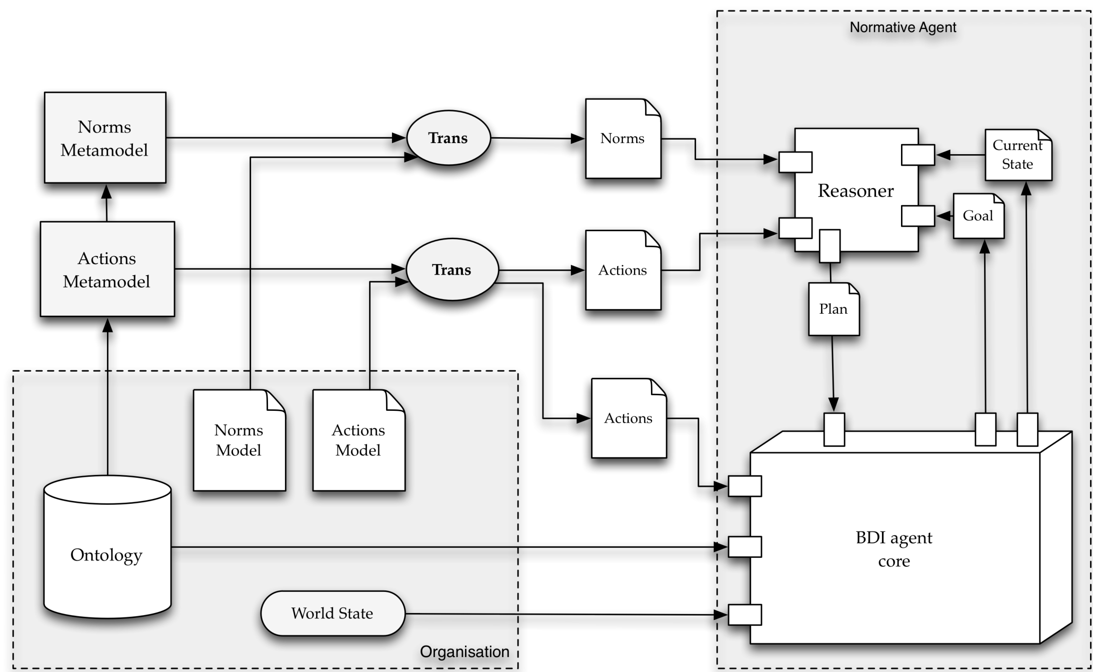


FIGURE 3.31: Framework architecture

The arrows in the diagram indicate inputs/outputs. The ellipses with the "*Trans*" label indicate model transformation functions. These functions receive a metamodel and a model (i.e. a metamodel instantiation) and produce domain descriptions appropriately adapted to the reasoner's domain representation language as well as to the agent's world description (it might be the case that the reasoner and the agent might have different representation languages for the world).

structure, work processes, and norms of the agent organisation in which it operates. As a result, the BDI agent processes and interprets (through organisational metamodels - in our case of actions and norms) the organisational descriptions (models from a repository) to produce its world state representation.

Before describing the normative reasoning component, it is important to note that there are in fact three types of normative reasoning involved in the BDI cycle.

1. Ontological and conceptual interpretation of the constitutive norms described in Section 2.2.2.2
2. Deliberative reasoning is done in order to select WHICH desires and intentions are to be interpreted as goals of the agent (see Section 3.3.2)
3. Means-ends reasoning, or what we otherwise call "norm-driven plan synthesis", is done in order to decide HOW to get things done within a normative environment, that is HOW to achieve the chosen goals

There have been several attempts to do all three types of reasoning with the same reasoner [Meneguzzi and Luck, 2009b; Meneguzzi et al., 2012; Cliffe et al., 2007b],

still, there are no optimal reasoners for this. Other approaches suggest the use of optimised reasoners for each one. As mentioned earlier, our thesis focuses on the relation between means-ends reasoning (that is the norm-driven planning procedure, which we refer to simply as a normative reasoner, where the course of action in order to achieve some goal is determined) and the deliberative reasoning.

That said, the reasoning component lies within the agent and needs to be able to perform normative reasoning while ensuring compliance with the organisational specification. More specifically, the reasoner receives a set of instantiated norms and a set of instantiated domain actions which conform to the norms and actions metamodels and, taking into consideration the agent objectives (goals) and the current state of affairs, produces a plan. The plan is valid and is considered to be the most beneficial, according to a given set of criteria.

3.3.2 BDI Agent Structure

In Section 2.1.1 the BDI (Belief-Desire-Intention) agent model was described. In this section we explain how we adopt and extend it in order to incorporate the normative reasoning process. According to Bratman [Bratman, 1987], the structure of a BDI agent comprises of:

- A set of *(B)eliefs*
- A set of *(D)esires* (or *goals*)
- A set of *(I)ntentions* - a subset of the goals
- A set of *perceptions* (or *events*)
- A *plan library* as a (static) component or a planning mechanism which produces plans.
- Perceptions are represented as new belief and new goal events (when agent observes something, is told something or is asked to do something)

On each cycle (see Algorithm 3.1) the set of perceptions is added to the set of internal events (plan generated sub-goal notifications) to give the events for that cycle. Then the process that takes place can be summarised as follows:

- The perceptions are what the agent reacts to taking into account its current beliefs, desires and intentions.
- The agent selects one pending perception.
- It responds by updating its beliefs, desires and intentions - it reconsiders existing goals and intentions.
- It reconsiders the plan to be followed.
- It then selects the next step in the plan being followed and, keeps executing the plan's actions, while revisiting the beliefs. An agent will maintain this process committing to its intention(s) until either a plan is completely executed, or the

Algorithm 3.1 BDI cycle [Wooldridge, 2001]

1: $B = B_0$
2: $I = I_0$
3: **while** true **do**
4: get next percept ρ
5: $B = brf^{a}(B,\rho)$
6: $D = options^{b}(B,I)$
7: $I = filter^{c}(B,D,I)$
8: $\pi^{d} = plan^{e}(B,I)$
9: **while** not ($empty(\pi)$ or $succeed(I,B)$ or $impossible(I,B)$) **do**
10: $\alpha = hd^{f}(\pi)$
11: $execute^{g}(\alpha)$
12: $\pi = tail^{h}(\pi)$
13: get next percept ρ
14: $B = brf(B,\rho)$
15: **if** $reconsider^{i}(I,B)$ **then**
16: $D = options(B,I)$
17: $I = filter(B,D,I)$
18: **end if**
19: **if** not $sound^{j}(\pi,I,B)$ **then**
20: $\pi = plan(B,I)$
21: **end if**
22: **end while**
23: **end while**

[a] *brf*(B,ρ) indicates a belief revision function. It revises its beliefs and updates them.
[b] *options*(B,I) is a function that generates a set of possible desires for the agent through its beliefs and intentions.
[c] *filter*(B,D,I) is a function that chooses between competing alternative intentions.
[d] π indicates a plan. It is be a sequence of actions $(a_1,\dots,a_n)$.
[e] the *plan*(B,I) function requires the agent to engage in a plan generation process. The plan generated (or selected in the case of using predefined plan libraries) satisfies the intention(s).
[f] the *hd*(π) function returns the first action of a plan.
[g] *execute*(α) is a function that executes an action in a plan.
[h] *tail*(π) function returns the plan minus the head of the plan.
[i] while an agent needs to stop and reconsider its intentions, reconsideration might prove costly. Therefore, the meta-level control component *reconsider*(I,B) is used to indicate whether or not to revisit the intentions of the agent.
[j] sound(π,I,B) function means that π is a correct plan for the agent's intentions given its beliefs.

agent believes it has achieved its current intention(s) or that its current intention(s) are no longer possible. If after revisiting its reconsidering its intention(s) a plan is not sound anymore, then the planning process comes up with a new plan to be followed.

- The agent then repeats the process to select another perception (if any).

Some theorists have suggested that obligation norms can be seen as a form of belief, and act within the process of generating (candidate) desires [Tufis and Ganascia, 2012] or can emphasise the importance of a desire in advance or define some priorities or precedence between desires (lines 6 and 16 in Algorithm 3.1). In a similar way, prohibitions might be considered to be another form of beliefs and should influence the process of filtering the desires in order to generate intentions (lines 7 and 17 in Algorithm 3.1).

In practice, the strategic selection of desires versus obligations and prohibitions is not as easy to do as it sounds in theory. While humans intuitively deliberate by projecting and assessing the options into the future and taking into account the obligations/prohibitions and their effects throughout the execution of the planning process, there is no such fixed way in which obligations and prohibitions can interfere with the selection of desires and intentions in a computing environment.

In [Meneguzzi et al., 2010], Meneguzzi et al. describe mechanisms that enable specific plan instantiations in order to restrict behaviour, support compliance and avoid violating plans. [Meneguzzi and Luck, 2009b] proposes a complex mechanism which, at runtime, reacts to newly accepted norms and constructs plans accordingly taking into consideration the intentions modified by the norms. Moreover, [Meneguzzi et al., 2012] evaluates plans with a preprocessing mechanism and annotates them depending on their degree of compliance to norms. Additionally, [Dignum et al., 2000] modifies the BDI cycle to incorporate again a preprocessing step for sophisticated plan selection, accommodating norms and obligations. Plans are explicitly expressed by the process of generating (candidate) intentions. Other researchers [Alechina et al., 2012; Dybalova et al., 2013] build on existing agent frameworks and allow the agent's deliberation cycle to be affected by the norms by adopting obligations as goals. However, while they calculate the normative state of the agent with respect to the norms during the cycle, they do not consider a planning methodology. Instead, they define preferences between obligations, prohibitions and goals, and they calculate algorithmically the optimal set of pre-fabricated plans (i.e. that are preference-maximal) to be executed.

The above works present good examples of the views of many theorists concerning the influence norms should have in the agent's deliberation cycle. They suggest that norms should influence the creation of intentions and/or modify the construction of the plans, based primarily on existing plan templates. An observation that can be made is that a lot of this research is also based on the preprocessing of possible plans in order to "speculate" to which extent the plan outcome will be compatible with

existing norms (or norms that might occur) and to decide whether an agent should or should not adopt it, even making comparisons between multiple possible plans in order to decide the course of action.

We, on the other hand, suggest that norms should be directly part of the planning process. That is, we think that norms should not interfere with the agent's objectives. On the other hand, they should act as restrictive constraints during the planning process, allowing to break some of those constraints in cases of conflict or whenever the agent sees fit for personal benefit. We believe this is better because, from a computational perspective, the preprocessing of partially exploring possible outcomes to select/prioritise intentions would consist of a similar mechanism to the one used during the means-ends reasoning, and thus, it makes little sense to do this exploration twice. The main effect of our approach is that the intention selection occurs during the planning process, in a way that: 1) candidate intentions which are infeasible are automatically discarded (until an opportunity makes them feasible again), 2) candidate intentions supported by obligations are more likely to be selected, and 3) candidate intentions hindered by prohibitions are less likely to be selected. Also, this approach provides a clear semantic distinction between the agent's objectives and the normative influence over these objectives during intention selection.

Algorithm 3.2 Modified norm-aware BDI cycle

1: $B = B_0$
2: ~~$I = I_0$~~
3: **while** true **do**
4: get next percept ρ
5: $B = brf(B,\rho)$
6: $D =$***options(B, D)***
7: ~~$I = filter(B,D,I)$~~
8: $\pi =$***plan(B,D,Ns,Pref)***
9: **while** not ($empty(\pi)$ ***or forall d in D: [succeeded(d,B) or impossible(d,B)]***) **do**
10: $\alpha = hd(\pi)$
11: $execute(\alpha)$
12: $\pi = tail(\pi)$
13: get next percept ρ
14: $B = brf(B,\rho)$
15: **if** ***reconsider(D,B)*** **then**
16: $D =$***options(B, D)***
17: ~~$I = filter(B,D,I)$~~
18: **end if**
19: **if** not ***sound(π,D,B)*** **then**
20: $\pi =$***plan(B,D,Ns,Pref)***
21: **end if**
22: **end while**
23: **end while**

The normative reasoning BDI agent used in our architecture includes the traditional components of a BDI agent, Beliefs and Desires, but leaves out the Intentions. Desires

are considered to be the objectives (goals) of the agent and the filtering process of the intentions is therefore eliminated (lines 2, 7 and 17 in Algorithm 3.2), leaving the means-ends reasoner free to consider all desires as goals to be achieved at some stage. The reasoner is integrated into the agent's BDI cycle as follows. Initially the reasoner produces a plan to be followed in order to achieve the objectives (line 8 in Algorithm 3.2). In our modified cycle, the planning process takes indirectly into account the existing norms and the agent's preferences, that are modelled as costs within the planning domain. While there is no external disturbance or information that changes the environment (perceptions) the agent executes actions of the plan. The norms' status is derived here from the current set of beliefs (lines 5 and 14 in Algorithm 3.2).

As explained, in the original BDI cycle of Algorithm 3.1 an agent gets to reconsider its intentions when either a plan is completely executed, or the agent believes it has achieved its current intention(s) or that its current intention(s) are no longer possible. Since in our suggested modification intentions no longer exist, the agent's commitment now is towards each of its desire(s) (line 9 of Algorithm 3.2), which are now directly considered as the agent's objectives to be accomplished.

The belief revision process in the agent is additionally affected by the execution of actions and the combination of their effects. Whenever there is a modification of the environment, the reasoner reproduces a plan according to the new state of affairs and passes it to the agent. Thus, the reasoner is applied in the BDI cycle whenever new planning is required. The main extension of the BDI process in our case lies in the fact that the planning process takes into consideration not only the beliefs but the norms that affect the agent as well. That is, norm influence is directly explored during the planning process (lines 8 and 20 in Algorithm 3.2). How this is done will be thoroughly explained in the rest of the thesis.

3.4 Norm Design

Norms lie at the heart of a normative environment, and their correct design is thus critical to the success of a norm-based application. We are interested in reasoning about the status of norms and normative environments, which we are unable to do with the machinery defined to this point. We therefore now provide a semantics for norms and normative environments, which allows us to determine the status of these entities at any point in time. Before doing so, we discuss some of the issues that occur and suggest a number of patterns that commonly arise when trying to model norms.

3.4.1 Norm Lifecycle and Norm Instances

In our framework, a norm might be in different states of *validity* within a normative system (see Figure 3.32): a norm is *in force* when it can be fully activated, monitored,

and enforced; *in transition* when it is being removed and cannot be activated anymore, but the effects of past activations have to be tracked until their end; and *deleted* when the history of the norm is to be kept but it can have no further effect on the normative system. Nevertheless, such a lifecycle is related to the concepts of promulgation, abrogation and derogation, and therefore out of the scope of this thesis.

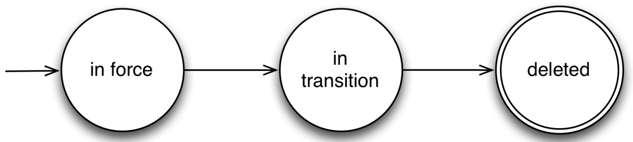


FIGURE 3.32: Norm lifecycle[19]

An issue that is very relevant to this thesis is the formal description of the *compliance* of a norm. In order to further investigate into this, it is necessary to make a distinction between a norm and its norm instantiations. This issue, i.e. a clear separation between an (abstract) norm and a particular (contextual) instantiation of the norm, is somehow missing in general in the literature. This problem was already discussed by Abrahams and Bacon in [Abrahams and Bacon, 2002]: *"since propositions about norms are derived from the norms themselves, invalid or misleading inferences will result if we deal merely with the propositions rather than with the identified norms*[20] *that make those propositions true or false"*. This issue is not banal, as it has implications on the operational level: in order to properly check norm compliance, norm instantiations and their lifecycle have to be tracked in an individual manner, case by case.

Therefore, to check a norm's compliance, it is necessary to have a link between the compliance of such instantiations and the norm. In most of the traditional formal foundations of deontic logics (such as in SDL), the *compliance* of a norm can be seen as binary: either the norm is fulfilled or is violated. The fact that, as we propose, a norm can be the origin – sort of a *template* – of potentially infinite instantiations, depending on the parameters that may activate it or the agents enacting its target role, can make the concept of norm compliance more complex.

Depending on the normative framework, there can be strict cases in which norm compliance has to be a binary value [Aldewereld et al., 2006], as well as there can be more relaxed scenarios in which norm compliance will completely depend on parallel concepts such as justice or fairness [Penner, 1988]. It is not the purpose of our formalisation to engage in such digressions, and thus we will make the simplest assumption: that *norm compliance is binary and depends on all of its instantiations*.

The concept of norm instance lifecycle has been treated by different authors, e.g. [Abrahams and Bacon, 2002; Fornara and Colombetti, 2009; Cardoso and Oliveira,

[19]This diagram, as well as ones that will appear later in the thesis, follows the standard notation of the state diagrams, where the arrow pointing from anywhere indicates an initial state and a double circle indicates a final state.

[20]From now on, we will denote such identified norms as *norm instances*.

2010], but with no real consensus. Taking those interesting elements that would allow the management of norms with the concepts of activation, maintenance, fulfilment and reparation, a suitable norm lifecycle would be similar to the one based on the automaton depicted in Figure 3.33. A norm instance gets activated due to a certain activating condition and starts in an (A)ctive state, but if at some point a certain maintenance condition is not fulfilled, the norm instance gets into a (V)iolation state. If the norm instance is (A)ctive and a certain discharge condition is achieved, the norm gets (D)ischarged[21]. Usually reparations are not treated explicitly, but in our proposal we add the concept for completeness. If a norm instance is (V)iolated, fulfilling a reparation condition can bring it back to the (A)ctive state, but if the discharge condition occurs while violated, only by fulfilling the same reparation condition (VD state) can the norm instance be (D)ischarged. It might be the case that a (V)iolated norm instance never gets repaired, so for safety we use a *timeout* condition[22] to make sure the norm instance is not alive forever and thus mark those permanent violations as (F)ailures.

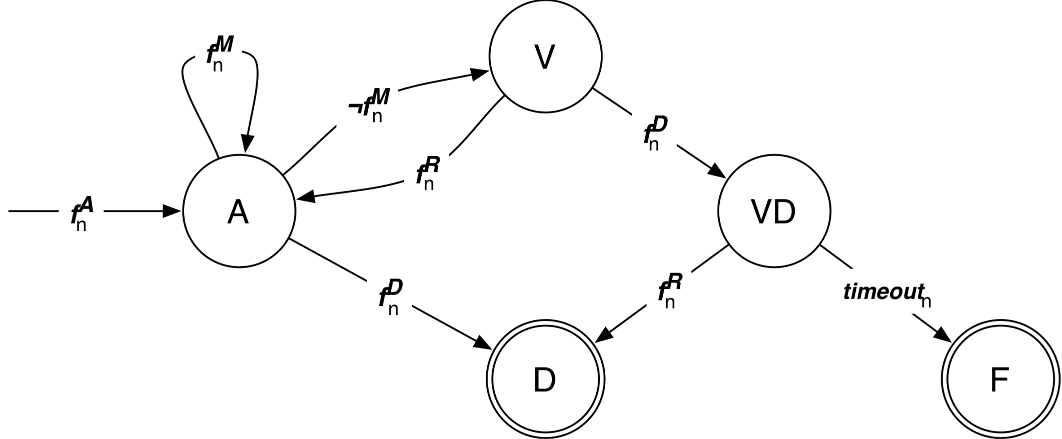


FIGURE 3.33: Norm instance lifecycle with reparation and timeout handling

Once there is a norm lifecycle the question to answer is how to deal with it from an operational perspective. Abrahams and Bacon [Abrahams and Bacon, 2002] solve this problem by considering instances as *occurrences* of the predicates contained in the deontic operator:

```
being_obliged_function_1
domain: X = any occurrence of borrowing from library
range:  being_obliged_function_1(X) where
             obliged first occurrence, on or after date of X,
             and within 14 days of X,
             of borrower in X returning borrowed in X
             (to lender in X)
```

[21]Note here that we assume the discharge condition to eventually happen.
[22]The timeout condition is evaluated as starting at the point of time of violation.

However, there are cases in which this can be insufficient, i.e. when the obligation defines a complex deadline or its instantiation depends on conditions based on contextual information. More recently, some works have been advancing towards tackling this issue. For example, by treating instantiated deontic statements as first-class objects of a rule-based language [Governatori, 2005; Cardoso and Oliveira, 2009], e.g.:

$$
\begin{aligned}
N_{top} = & ifact^{X:sa}(order(A1, Res, Qt, A2), T)^{23} \wedge \\
& supply\text{-}info^{X:sa}(A2, Res, Upr) \\
& \rightarrow \\
& obl^{X:sa}(A2, delivery(A2, Res, Qt, A1), T+2) \wedge \\
& obl^{X:sa}(A1, payment(A1, Qt * Upr, A2), T+2)
\end{aligned}
$$

In this example, taken from [Cardoso and Oliveira, 2009], we can see how the obligation (the deontic statement) is directly used in the right hand side of a production rule. This norm is read as: *for any supply agreement, when an order that corresponds to the supply information of the receiver is made, he is obliged to deliver the requested goods and the sender is obliged to make the associated payment*.

However, as these deontic statements are already implicitly identifying the norm instance through the variables inside the predicates and time T, there is no explicit tracking of which elements of the domain are involved in fulfilling or violating it. Other approaches declare the norm only at an abstract level while keeping track of its instantiations and its lifecycle at an operational level [Criado et al., 2010b; Álvarez-Napagao et al., 2010], e.g.:

```
rule ``norm instance fulfilment''
when
  Deactivation(n :  norm, f :  formula)
  ni :  Instantiated(norm == n, theta :  substitution)
  SubsetEQ(theta2 :  subset, superset == theta)
  Holds(formula == f, substitution == theta2)
then
  retract(ni);
  insert(new Fulfilled(n, theta));
end
```

Finally, some authors mention but do not handle the notion of norm instances in their frameworks [Meneguzzi and Luck, 2009b; Meneguzzi et al., 2010].

Returning to our framework design, as mentioned earlier in this section, we adopt the view that norm compliance is binary and depends on all of its norm instances.

[23] $ifact^{C}(f,t)$ says that institutional fact f has occurred at time t. Context identifier C can be seen as as a pair *id:type*, where *id* is a context identifier and *type* is a predefined context type.

That suggests that in order for a norm to be considered fulfilled, all the instances that occur throughout an execution must be fulfilled, each one individually. We consider our approach towards the norm instance definition to be an extension to the work in [Álvarez-Napagao et al., 2010]. Instantiations occur whenever there is an incidence of a substitution, sufficient to make the activating condition of the norm true (and therefore giving rise to a new instantiation). We explain more and provide a formal definition of norm instances, their lifecycle, norm instance fulfilment as well as norm fulfilment in Sections 4.3.3 and 4.3.4.

3.4.2 Dynamics of Norms

A temporal environment records the time at which some state change occurred within the environment, and the changes that took place. We assume that if a state change is not recorded between two time points, no change took place.

Philosophically, we adopt the stance that a norm does not have to be intentional. As long as a state of affairs referred to in an obligation/prohibition/permission occurs, regardless of whether the agent intended it to, the agent has met/violated/made use of the norm. This approach is based on the human model of evaluating evidence in the eyes of the law.

When performing verification, we may assume a fully observable domain, and make use of a single theory. For the purposes of monitoring, the agreed theory is the one that is shared among the monitoring components. By contrast, when performing practical reasoning, individual agents may make use of their own theories.

3.4.3 Primary, Secondary Norms and Interaction Between Them

We have already defined an optional extension to the framework where we distinguish between *primary norms* (which are the ones that maintain the essence of a rule) and *secondary* (or *repair*) ones, which are the ones that get triggered by the violation of a primary norm. The second type of norm is modelled by including the "violation" status of the primary norm in the activating condition of the secondary norm. We use this extension in Chapter 5. Due to the free nature of this representation, it is possible to have a full structure of norms starting with a primary one and each of the rest serving as repair norms for the previous one.

Our normative model focuses on the status of single norms. We assume that no other (to the one explained above) semantic interaction exists between norms and that each one represents a semantically separate deontic statement within the environment. Therefore, while our framework can assign a status (such as violated) to a norm, it does not perform reasoning about the interactions between norms. Instead, it might include the conceptual tools for other components to use when reasoning about norm

interactions. Thus, while we facilitate more sophisticated reasoning about norms, such reasoning is not addressed in the core framework.

3.5 Discussion

The work presented in this chapter has been developed within (and highly shaped by) the IST-CONTRACT and ICT-ALIVE [Lam et al., 2009] projects[24]. The first offers a multi-layered contracting framework, its architectural components and tools. The normative environment formalisation of Appendix A, that stood as inspiration to the thesis semantics described in Chapters 4 and 5, has been used in the project to model the contracts established between agents as sets of clauses (representing deontic statements) and the message interaction between them. It also formed a common baseline for the project partners to develop local and global monitoring tools as well as verification components. Additionally, as part of the project, contract-supporting components like contract editors, a contract store and contract managers were developed. ICT-ALIVE's objective, on the other hand, was to merge previous work in coordination and organisational structures with the most up-to-date technology in service-oriented computing, allowing system designers to create service-oriented systems based on the organisational structures and the way these communicate between them. The idea for decision making through planning mechanisms presented in Chapters 4 and 5 started during the ICT-ALIVE project but has evolved beyond the project scope and lifetime.

In this chapter we have made a detailed requirements analysis, deriving directly from the problem definition for this thesis in Section 1.1, over the reasoning mechanism required when dealing with norm-driven agents. Based on these requirements, we created a computational framework, comprising of the basic elements used when designing such a mechanism.

We picked the MDE approach, which separates the development of software systems into different, independent phases, where the top one remains an abstract representation of a platform-neutral model of the system. In this way we manage to raise a level of abstraction between our framework elements and the two implementation approaches presented in Chapters 4 and 5 respectively. The conceptual elements in our framework (agent, plan, agent capabilities, etc.) are elements that are popularly used in many theoretical and practical frameworks (e.g. in Jason [Bordini and Hübner, 2006], Jadex [Braubach et al., 2005], JACK [Winikoff, 2005] and 2APL [Dastani, 2008]), making it easy as we will see in Chapter 5 to create and apply automatic translations and transformations from our models to various platform-dependent models.

Our architecture places the normative reasoning component inside a BDI agent operating in a wider environment. A complete integration within such a multi-agent

[24]See Section 2.2.2.3.6 for a description of IST-CONTRACT and Section 2.2.2.3.3 for a description of ICT-ALIVE

environment will be provided in Chapter 5. To our knowledge, only recently has the community started getting interested in BDI-oriented agent reasoning with norms (see Section 2.2.2.4.2 for relevant work) and relatively little work on how the agent's cycle can be modified to include normative influences throughout the means-ends reasoning has been done.

In addition to the above, we have in this chapter introduced the basic design issues when introducing norms in normative environments (seen as a special type of institution) and provided a first glimpse of the norms' representation on an operational level. While a rich background on norm representation exists on a conceptual level, fewer researchers attempt to define and resolve issues such as multiple instantiations of the same norm, the flexibility that a norm might provide for a possible breach and the handling of the consequences of such an event on a practical level. In Chapters 4 and 5 we address these issues with success, providing a pragmatic aspect of how norms affect and interfere with the decisions to be made by a goal-driven agent.

Since our general interest is that an agent is able to exist and make decisions in normative environments, guided by a sophisticated reasoning mechanism which allows for runtime decisions to be made, the elements of our framework should be assigned operational semantics taking into account the effect the norms have on the agent's status and the environment. In order to do so, we need to evolve the formalisation described here to additional levels, where actions, norms, states, goals and plans acquire operational semantics. These extensions of the formalisation and its semantics are covered in the next two chapters, through two different approaches.

Chapter 4

Normative Practical Reasoning Using Temporal Control Rules

As explained in Section 1.1 one of the main constraints in the use of deontic-like formalisations is the lack of operational information [Vázquez-Salceda and Dignum, 2003] that might make it easy to represent them in computational terms and apply in computational systems. Without such semantics, an agent cannot clearly determine what the influence of norms might be on its decision making process.

As detailed in Chapter 2, in the literature there is a lot of work on normative systems' formalisations (mainly focused on deontic-like formalisms [von Wright, 1951]) which are declarative in nature, focused on the expressivity of the norms [Dignum et al., 2005], the construction of formal semantics [Boella and van der Torre, 2004; Aldewereld et al., 2006; García-Camino et al., 2006] and the verification of consistency of a given set [Lomuscio et al., 2009; Governatori and Rotolo, 2010]. However, often such formalisations cannot really be used at execution time. Also, there is some work on how agents might take norms into account when reasoning [López y López et al., 2004; Kollingbaum, 2005; Meneguzzi and Luck, 2009b], but few practical implementations exist that cover the full BDI cycle, as many approaches do not include the *means-ends reasoning step*.

As a consequence, there are several critical points that are not fully covered, to our knowledge, in any work in the literature. In scenarios where individual agent reasoning takes place, the following are such relevant issues that should be taken into consideration:

- Practical implementations define their own operational semantics in order to bring the abstraction of the concept of norm to the agents' reasoning. In most cases, this is done by defining semantics close to the implementation languages loosely based on, and therefore similar but not reducible to, deontic logics. In such cases, there is always the risk of 1) being misaligned with the philosophical foundations of normative systems, and 2) not being able to compare, from a scientific point of view, different proposals that are supposed to tackle the same

domain of problems. In summary, *there is a need to formally connect the deontic aspects of norms with their operationalisation, preserving the former*.

- From a practical point of view, *norms have to be distinguished from their actual instantiations. For each norm, many instantiations may happen during the norm's lifetime*[1].
- Normative systems can be tailored for different purposes. Although there is a strong focus in this document on how to deal with norms in agents' individual reasoning from a planning perspective, other systems may deal with other aspects, such as normative monitoring, norm compliance predictive systems, model checking of institutional properties, and so on. Each of these system objectives may imply completely different implementation languages and/or algorithmic complexity. In many cases, one would also want to combine several of these techniques in order to build a tailored system for a particular domain solution. For this respect, *the operational semantics should ideally be formalised in a way that ensures enough flexibility in their translation to actual implementations while ensuring unambiguous interpretations of the norms*. For instance, the semantics used by a society compliance mechanism, and the semantics integrated in the reasoning cycle of its individual agents, must be aligned to avoid, e.g. the agent norm reasoning mechanism stating that no norm has been violated while the compliance mechanism states that the agent has violated some norm instance.

Inspired by our previous work on norm lifecycle semantics[2], we take it a step further and suggest that decision making in a normative environment can be implemented via a planning mechanism. The mechanism, integrated into an agent's deliberation cycle, produces and at the same time evaluates the plans[3]. In possession of a set of norms which can be seen as indications over desired behaviour of the agent and an objective, and taking into account the current state of affairs, the agent computes the most beneficial way to achieve its objective, that is, what sequence of actions should be followed in order to reach it, gaining at the same time optimal cost/benefit though fulfilling or ignoring obligations and prohibitions.

An important contribution of our work is the introduction, semantic representation and handling of norm instances. As mentioned in Section 3.4.1, the problem of separating a norm from its instantiations has been tackled from a theoretical perspective and few works actually include and create full sets of norm instances, independent at execution time. In this chapter we bring norm instances down to a practical level (Section 4.3) and demonstrate how to include them in the decision process of the agent (Section 4.6).

[1] Please refer to Section 3.4.1 for the motivation of this desirable property.

[2] The formalisation in [Oren et al., 2009] was a first attempt made to modelling the operational semantics of regulative norm instances. Although the formalisation is incomplete and has been superseded by later attempts, it is described in Appendix A, as it has inspired our current formalisation.

[3] The integration to the deliberation cycle will be shown in Section 5.3.

To perform norm-aware planning, we present a proposal to achieve a deontic-based norm definition with a direct reduction to temporal logic which, in turn, can be translated into planning operational semantics. Choosing the way to represent and formalise the norms depends on the aspects of the norms that are of great significance for the design of a regulated system. While the formalism should capture the deontic notions expressed in the norms, this is not sufficient as information on the temporal ordering of the normative states of the norms is missing when trying to establish the necessary temporal timeline that indicates how a norm evolves and gets fulfilled (or not) throughout time. The formalism picked should additionally be expressive enough to allow a temporal aspect, while at the same time including an indirect notion of actions, necessary when dealing with agents that engage in activities and evolve throughout time, and the states these bring about. For these reasons, the use of temporal logic seems most appropriate considering its expressiveness when dealing with the concept of time and conditional satisfaction of state of affairs. We have in particular chosen Linear Temporal Logic (LTL) as a bridge from the norm specification to its implementation by reducing deontic-based norm definitions to temporal logic formulas which, in turn, can be translated into planning semantics. The reason for picking LTL is that it has operational semantics that can be directly implemented in computational systems while, at the same, time many algorithms have been developed and integrated into verifying and model checking systems. LTL also appears to be the preferred language when expressing controlled search within domain-specific knowledge and in this way we take advantage of existing, stable frameworks in the planning community that use LTL formulas as strong and soft constraints on plan trajectories (e.g. TLPLAN [Bacchus and Kabanza, 2000] and PDDL 3.0 [Gerevini and Long, 2006]).

In short, there are many approaches that tackle different parts of the formalisation of norm operationalisation. One of the purposes of this chapter is, thus, to complement these approaches by filling the gaps that exist between the deontic statements and planning operationalisation by means of 1) additional predicates representing norm activation, discharge and violation, and 2) an intermediate representation based on temporal logic (see Section 4.3).

4.1 First-Order Linear Temporal Logic

Let us first define the formal foundations for our formalisation. The *first-order Linear Temporal Logic* (*fo-LTL*) language starts with a standard first-order language, $\mathcal{L}$, containing some collection of constant, function, and predicate symbols, along with a collection of variables (we use the framework elements of the logic defined in Section 3.2.1 to form formulas in $\mathcal{L}$). We also include in the language the propositional constants $\top$ and $\bot$ (true and false respectively), which are treated as atomic formulas.

- if $f_1 \in \mathcal{L}$ then f_1 is a fo-LTL formula;
- If f_1 and f_2 are fo-LTL formulas, then so are $f_1 \mathbf{U} f_2$, $\mathbf{X} f_1$, $\mathbf{F} f_1$ and $\mathbf{G} f_1$

We define a substitution (grounding) $\theta = \{x_1 \leftarrow t_1, x_2 \leftarrow t_2, ..., x_i \leftarrow t_i\}$ as the substitution of the terms $t_1, t_2, ..., t_i$ for variables $x_1, x_2, ..., x_i$ in a formula $f \in \mathcal{L}$.

A full path $\pi = < s_0, s_1, s_2, \ldots >$ is a sequence of *states of the world*. Every state of the world (or state) is a model for the base first-order logic $\mathcal{L}$. What is more, every state s_i shares the same domain of discourse D.

Definition 4.1. Validity of an fo-LTL formula over a path π is defined as:

(a) If f_1 is an atomic formula then $\langle \pi, i, V \rangle \models f_1$ iff $(s_i, V) \models f_1$. That is, atomic formulas are interpreted in the state s_i under the variable assignment V according to the standard interpretation rules for first-order logic.

(b) $\langle \pi, i, V \rangle \models \forall x. f_1$ iff $\langle \pi, i, V(x/d) \rangle \models f_1$ for all $d \in D$ where $V(x/d)$ is a variable assignment function identical to V except that it maps x to d.

(c) $\langle \pi, i, V \rangle \models f_1 \mathbf{U} f_2$ iff there exists $k \geq i$ such that $\langle \pi, k, V \rangle \models f_2$ and for all j with $i \leq j < k$ we have $\langle \pi, j, V \rangle \models f_1$.

(d) $\langle \pi, i, V \rangle \models \mathbf{X} f_1$ iff $\langle \pi, i+1, V \rangle \models f_1$

(e) $\langle \pi, i, V \rangle \models \mathbf{F} f_1$ iff there exists $j \geq i$ such that $\langle \pi, j, V \rangle \models f_1$

(f) $\langle \pi, i, V \rangle \models \mathbf{G} f_1$ iff for all $j \geq i$ we have $\langle \pi, j, V \rangle \models f_1$

Additionally, we define the following:

(g) $\langle \pi, i, V \rangle \models \exists x. f_1$ iff $\langle \pi, i, V(x/d) \rangle \models f_1$ for some $d \in D$.

(h) $\langle \pi, i, V \rangle \models f_1 \vee f_2$ iff $\langle \pi, i, V \rangle \models f_1 \vee \langle \pi, i, V \rangle \models f_2$

(i) $\langle \pi, i, V \rangle \models f_1 \wedge f_2$ iff $\langle \pi, i, V \rangle \models f_1 \wedge \langle \pi, i, V \rangle \models f_2$

More details about first-order Linear Temporal Logic can be found in [Bacchus and Kabanza, 2000].

4.2 Extensions of fo-LTL for norms

fo-LTL is a logic that is tailored for generic handling of paths of action. In order to properly work in the domain of norms, there are two issues that need tackling: *time*, in order to handle temporal intervals that define deadlines, discharge conditions or sanction timeouts; and *agency*, in order to bind obligations, permissions and prohibitions with the agents responsible to bring them about.

We handle time in a trivial manner, defining discrete time steps corresponding to the sequential states of a path[4]. That allows us to create a simple notion of time starting at 0, which would correspond to the initial state of the world, and for every state of a

[4]Alternatively, other time functions or approaches can be considered, according to the designer's domain implementation and preferences.

path, time augments by one. While this is a uncomplicated way of considering such a complex subject, it allows for time to be easily introduced and implemented in almost all practical frameworks[5].

Further to the notion of time, we need to be able to describe relative times in our model, since we will want to state that norms get repaired within a time limit from the occurrence of a violation. Therefore, an absolute model of linear time, counting from the beginning of an execution is not sufficient. We borrow the notion of quantitative relationships for linear temporal logic from other logicians [Bellini et al., 2000; Koymans, 1990] but do not use the full semantics of logics such as Metric Temporal Logic (MTL) [Koymans, 1990], since we do not consider a time model in our framework. Therefore, we introduce the symbol $_\leq$ in combination with the operator **F**, written as $\mathbf{F}_{\leq t}\phi$ to indicate that "*eventually, within t time steps, ϕ will be true*". Such an inclusion does not modify any of the LTL properties, since $\mathbf{F}_{\leq t}\phi$ can be seen as an operator weaker and more constrained than the operator F. Consequently, we extend fo-LTL to contain a relative eventually operator:

(j) $\langle \pi, i, V \rangle \models \mathbf{F}_{\leq t} f_1$ iff there exists j with $i \leq j \leq i + t$ such that $\langle \pi, j, V \rangle \models f_1$

Another issue to take into consideration is that the axiomatic definition of the semantics of fo-LTL – and, for that, matter, of other temporal logics such as LTL, CTL or CTL* – does not explicitly include a reference to the concept of agent. Therefore, with the semantics as they are, there can be no binding between the responsibility of achieving or maintaining a certain state and the actual accomplishment of such responsibility. In the context of any work on normative systems, being able to reason about this binding is a necessity. There is a strong research background [Belnap and Perloff, 1988; Horty, 2001; Bentzen, 2010] on this topic.

In order to deal with that, we extend fo-LTL and add a notation that will provide semantics for *agency*, based on the *stit (see to it that)* operator, firstly introduced by Belnap and Perloff in [Belnap and Perloff, 1988], with $E_\alpha[\rho]$[6] indicating that "*agent α sees to it that ρ happens*", i.e. the truth of ρ is guaranteed by an action performed by α. We axiomatise our operator by stating that in our setting, performing a stit action is a one time-step procedure and therefore our operator obeys $\langle \pi, i, V \rangle \models E_\alpha[\rho] \Rightarrow \langle \pi, i, V \rangle \models \rho$.

In order to have formal semantics for such an operator, a logical binding between 1) the agent who *sees to it that* ρ happens and 2) the agent who can *choose* among available actions at any moment in a setting and actually *produces* the set of actions needed to realise such a fulfilment, is essential. As we will again see later in Section 4.3.3, it is not in the scope of this thesis to provide specification and operationalisation of such

[5] In this thesis, we do not investigate the translation of continuous, real time, deadlines and complex time intervals towards our simplistic model of time. While it is an interesting topic, representation of time in reasoning models has been covered in different works [Fisher, 2008].

[6] First representations of the stit logic use the notation $[\alpha\ stit : \rho]$ or $[\alpha\ stit]\rho$ but we find $E_\alpha[\rho]$, defined in [Dignum et al., 2005], more elegant and practical for our purposes.

a formal connection and keep the simple assumption that an agent automatically is responsible for the situation to be brought about.

4.3 Formalisation

In this section, we present a proposal for a deontic logic for support for norm instantiation via obligations parametrised by four states (conditions)[7].

Before proceeding with the formalisation of our concepts, we introduce the *normative model*. In order to be able to talk about normative reasoning performed by agents, we need to place this inside a normative context. As described in Section 3.2, our conceptual framework is an abstraction of elements needed to perform normative reasoning. Therefore, by the term normative model we refer to an instance of our **framework:NormativeModel**, as defined in Section 3.2. In the rest of the document we assume the existence of such a normative model, denoting it as NM where necessary.

Definition 4.2. A Normative Model is defined as:

$$NM = \langle Roles, Agents, s_0, Actions, Norms, Context \rangle$$

where $Roles$ is a set of roles, $Agents$ is a set of agents, s_0 is the current system state, $Actions$ is a set of actions, $Norms$ is a set of norms and $Context$ is the framework context, with all these elements as defined by the conceptual framework explained in Section 3.2.

Furthermore, for the purpose of this formalisation, we assume the use of a first-order language $\mathcal{L}$ as in Section 4.1. We also adopt the notion of state from the same section.

4.3.1 Norms

In this chapter we formally define a *norm*, adding elements for tracking of reparation of violations.

Definition 4.3. We define a norm n as a tuple $n = \langle r, f_n^A, f_n^M, f_n^D, f_n^R, timeout_n \rangle$, where:

- $\boldsymbol{r}$ is the agent role the norm applies to (i.e. only agents enacting the role r will be obliged to comply with instances of this norm).
- $\boldsymbol{f_n^A}$ is the activating condition of the norm
- $\boldsymbol{f_n^M}$ is the maintenance condition of the norm
- $\boldsymbol{f_n^D}$ is the discharge condition of the norm

[7] These conditions have been already in our framework (see Sections 3.2.6, 3.4) as well as in our previous work on norm semantics inspiring our current formalisation, found in Appendix A.

- $\boldsymbol{f_n^R}$ is the repair condition of the norm
- ***timeout$_n$*** is an optional number (considered ∞ in case it is absent) that represents the upper-bound time limit for the reparation of a violation, taken into account only after a violation and not before, and
- $f_n^A, f_n^M, f_n^D, f_n^R \in \mathcal{L}$

If, for example, we wanted to model the following norm applying to all agents enacting the citizen role Cz: "*While Cz is driving, he is obliged to not during a red traffic light, otherwise he will have to pay a fine with cost 100 before 50 units of time pass.*", the result would be:

$n = \langle Cz, \{driving(Cz)\}, \{\neg crossed\text{-}red(Cz, L)\}, \{\neg driving(Cz)\}, \{fine\text{-}paid^{8}(100)\}, 50\rangle$

In order to convert this normative tuple into a deontic statement, Standard Deontic Logic is insufficient. First of all, SDL – in its form $O(A)$ – does not cover the possibility of having conditional statements that determine when a norm starts being in force, an issue that is tackled by Dyadic Deontic Logic [Prakken and Sergot, 1997] – $O(A|B)$: given B, A should happen. On the other hand, as seen in Section 3.4.1, our norm lifecycle allows for the definition of checkpoints that indicate a normative discharge or a timeout condition. In order to deal with them, we take inspiration from the semantics of deadlines in [Dignum et al., 2005].

The deontic interpretation of the tuple in Definition 4.3 is done by means of the deontic formula.

Definition 4.4. Given a normative model $NM = \langle Roles, Agents, s_0, Actions, Norms, Context\rangle$, the deontic interpretation of a norm $n = \langle r, f_n^A, f_n^M, f_n^D, f_n^R, timeout_n\rangle$ belonging to $Norms$, with agent α in $Agents$, role r in $Roles$, such that $enacts(\alpha, r)$ is:

$$\vdash_{NM} O_{f_n^R \leq timeout_n}(E_\alpha[f_n^M] \preceq f_n^D \mid f_n^A)$$

The syntax of the operator proposed is similar to the obligation operator from the semantics of the logics aforementioned: Dyadic Deontic Logic and semantics of deadlines. However, there are some notable differences. While the $\leq$ used for $f_n^R \leq timeout_n$ corresponds to the deadline semantics [Dignum et al., 2005] (if $timeout_n$ occurs, there is a permanent violation), the $\preceq$ used in $E_\alpha[f_n^M] \preceq f_n^D$ should rather be read as "f_n^M should hold *at all times* at least until f_n^D". Also, the conditional notation $|$ used in Dyadic Deontic Logic, which does not always have clear semantics in terms of temporality, in the case of the operator proposed $O(A|B)$ should be read as "from the

[8] Beside the fact that each time there is an infraction the fine has to be paid, we use a predicate $fine_paid$ that keeps no track of the different violations. We do this in order to keep the domain design simple and avoid an excessive number of parameters. Alternatively, one could design the same predicate to include extra parameters such as the traffic light L in order to indicate each individual infraction.

moment B happens, A should happen" rather than simply "given B, A should happen"[9].

Therefore, the expression shown in Definition 4.4 is informally read as: "*if at some point f_n^A holds, agent α is obliged to see to it that f_n^M is maintained until, at least, f_n^D holds; otherwise, α is obliged to see to it that f_n^R holds before $timeout_n$*". Note that in this informal reading we are not dealing with norm instances yet. How we address this issue will be explained in Section 4.3.2 and full semantics for the way operator O gets satisfied with respect to the occurring norm instances will be given later, in Section 4.3.4. Following the example,

For every agent Ag such that $enacts(Ag,Cz)$:

$$\vdash_{NM} O_{fine\text{-}paid(100)\leq 50}(E_{Ag}[\neg crossed\text{-}red(Ag,L)] \preceq \neg driving(Ag) \mid driving(Ag))$$

informally read as: "*if at some point Ag is driving, Ag is obliged to see to it that no red light is crossed until, at least, Ag is not driving anymore; otherwise, Ag has to pay a fine of 100 before 50 units of time pass*". The semantics of this operator are presented in the rest of this section.

It is important to remind here that, within our framework, we can model both obligations and prohibitions, as prohibitions are modelled as a form of obligation (see Section 3.2.6). [10]

4.3.2 Norm Instances

Inspired from the discussion in Section 3.4.1, we have to address the following issues when dealing with norm modelling into account:

1. Deontic statements do not express truth values related to a norm, but rather the existence of a norm [Walter, 1996].
2. In order to check the compliance of a norm, its particular instances must be tracked [Abrahams and Bacon, 2002],

A norm is defined in an abstract manner, affecting all possible participants enacting a given role. In order to work with instances, we need to define a norm instantiation. We consider a substitution θ as defined in Section 4.1. Whenever a norm is active, we will say that there is a *norm instance* n_α^θ for a particular norm n, an agent α and a substitution θ.

[9] In some works in the literature, this is interpreted as "given B and as long as B happens, A should happen" (e.g. [Prakken and Sergot, 1997]), while in other works it is interpreted in a way closer to our reading (e.g. [von Wright, 1971]).

[10] In Section 4.3.2 we will actually see an example of prohibition modelled in our framework.

Definition 4.5. Given a normative model $NM = \langle Roles, Agents, IS, Actions, Norms, Context \rangle$ and a norm $n = \langle r, f_n^A, f_n^M, f_n^D, f_n^R, timeout_n \rangle$ in NM, with α in $Agents$, r in $Roles$, such that $enacts(\alpha, r)$ and a substitution θ, we define a norm instance n_α^θ to be $n_\alpha^\theta = \langle \alpha, r, \theta f_n^A, \theta f_n^M, \theta f_n^D, timeout_n \rangle$, where:

- θf_n^A is fully grounded,
- θf_n^M, θf_n^D may be fully or partially grounded and
- $timeout_n$ is a number representing the time limit

The reason that θf_n^M, θf_n^D may be partially grounded is that the substitution that instantiates the norm – that is, θ such that θf_n^A holds – is considered in our model to be the sufficient and necessary set of substitutions needed to fully ground f_n^A.[11]

4.3.3 Norm Lifecycle

Although LTL as a formalism is suitable enough in terms of complexity for reductions to monitoring and planning scenarios, and therefore for practical reasoning from an institutional or individual perspective, there are intrinsic constraints that limit the expressiveness of the framework.

More concretely, the norm instance lifecycle proposed in Figure 3.33 cannot be expressed in LTL. As proved in [Tauriainen, 2006], in order to reduce an automaton to an LTL expression – and vice versa –, it necessarily has to be a *self-loop alternating automaton*: free of loops that involve more than one state, i.e. only cycles that start and finish in the same state and involve no second state are allowed.

This is an important constraint that prevents our model from having a loop between the (A)ctive and the (V)iolation states. In other words, if we want to use LTL, the lifecycle cannot have cycles that allow it to go backwards. Therefore, for the purpose of our formalisation, we propose to adopt the more straightforward lifecycle shown in Figure 4.1.

The main difference with respect to the automaton in Figure 3.33 is the handling of violations. As there is no way back to an (A)ctive state anymore, from a (V)iolation state there are only two options: either to repair the norm instance and subsequently (D)ischarge it, or mark it as a (F)ailed if it has not been dealt with for a given amount of time. This raises the issue of the norm instance not being able to indicate a second violation while it is already violated, meaning that another violation cannot be observed while it is already in a violation state. From an operational perspective, this issue can be worked around by allowing the norm-aware system to create more instances of the same norm if an instance is violated before a discharge.

[11] It can be the case that the set of variables used in f_n^M and/or f_n^D is larger than the cardinality of θ though. Let us suppose, for example, that a norm is designed to be instantiated at all times while it is in force. In that case, regardless of any contextual condition, f_n^A will be set to $\top$.

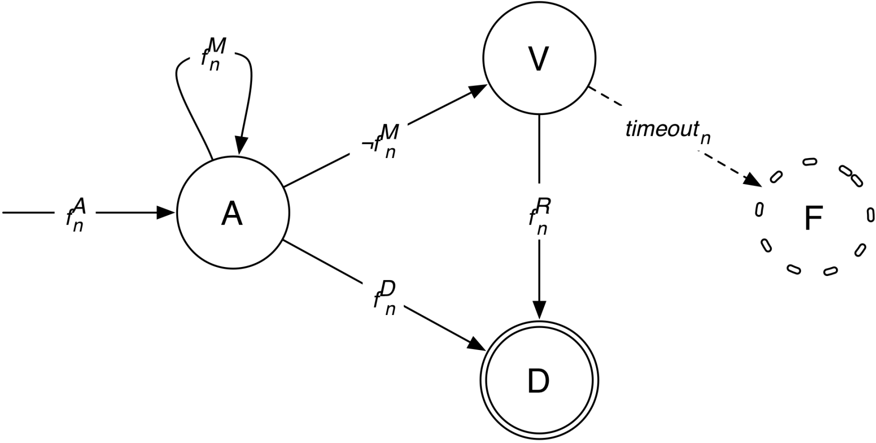


FIGURE 4.1: Self-loop alternating automaton-based norm instance lifecycle.

The arrow pointing from anywhere indicates an initial state and double circles represent final states. The non-dashed part of the figure indicates the states that we wish to model by formalising the norm lifecycle of an instance, while the dashed part indicates states that should not be reached during the norm lifecycle. In other words, the dashed part of the figure indicates the unwanted failed state of the norm instance.

For a norm to have a deontic effect, it is required that the activating condition actually happens at some future point. Additionally, either of the following three conditions should happen:

- The activating condition never occurs so the norm never gets activated.
- Always, between the activating and discharge condition, the maintenance holds (reached "discharged" state).
- The maintenance condition holds up to a point where it becomes false and then a violation is permanently raised. In addition, the repair condition occurs later (reached "discharged" state) before timeout is reached.

In this way we tie the maintenance of a condition to the violation of a norm and therefore, the deontic effect of a norm can be described by the causal effect between the maintenance condition and a violation in Definition 4.6.

Following what was discussed at the beginning of this section, deontic statements are substantive in the sense that they do not express anything further than the mere existence of a particular norm or, in our case, of a norm instance. Therefore, we have to distinguish between two statements that will co-exist in our semantics: 1) that a certain norm instance exists, and 2) that a certain norm instance has been correctly fulfilled/has failed. While the former refers to this substantive aspect, the latter provides a logical grounding in our model.

In order to give meaning to the substantive aspect, we define a specific operator $\mathcal{O}$ with similar syntax to the norm operator O, which indicates the existence of a norm

instance, an active instance produced from a norm *in force*. Additionally, we make use of the *Gödelisation* operator $\lceil . \rceil$ [Gödel, 1931] for naming norm instances in our language. That is, $\ulcorner n_\alpha^\theta \urcorner$ names the instance n_α^θ and allows us to use it in formulas. Let $\mathcal{L}$ be a standard first-order language for the formation of formulas, θ a variable substitution (as described in Section 4.1), $\pi =< s_0, s_1, s_2, \ldots >$ a sequence of states and $failed(\ulcorner n_\alpha^\theta \urcorner)$ a predicate belonging to $\mathcal{L}$ representing the failure of a norm instance n_α^θ, subsequently reaching the state (F).

We can now establish the semantic relationship between the lifecycle of a norm instance and the fulfilment/violation of a norm as in Definition 4.6:

Definition 4.6. Causal semantics for the operator $\mathcal{O}$

$$
\begin{array}{l}
\langle \pi, i, \theta \rangle \models \mathcal{O}_{\theta f_n^R \leq timeout_n}(E_\alpha[\theta f_n^M] \preceq \theta f_n^D \mid \theta f_n^A) \\
\quad \text{iff} \\
\langle \pi, i, \theta \rangle \models \\
\qquad \mathbf{G}(\neg f_n^A) \vee \\
\quad \Big[\neg f_n^A \mathbf{U} \Big(f_n^A \wedge [\forall \theta' : \theta' f_n^M \mathbf{U} \exists \theta'' : \theta'' \theta' f_n^D] \Big) \Big] \vee \\
\quad \Big[\neg f_n^A \mathbf{U} \Big(f_n^A \wedge [\exists \theta' : \theta' f_n^M \mathbf{U} (\neg \theta' f_n^M \wedge [\exists \theta'' : \mathbf{F}_{\leq timeout_n} \theta'' \theta' f_n^R])] \Big) \Big]
\end{array}
$$

Definition 4.6 reflects the non-dashed part of Figure 4.1. This is because (F) is an "unwanted" state and should be avoided when a norm is in force. The first line of the temporal formula says that the activating condition actually never happens and the execution does not incur in a violation throughout the execution path. This case does not cause any change in the state of the system. The second line says that there exists some substitution for the activating condition in the future (A), and that always until a substitution raises an instance of the discharge condition, the maintenance condition holds for all substitutions. No violation occurs throughout the execution path. This case terminates the norm in a state of discharge (D). The rest of the lines in the formula imply that there exists some substitution for the activating condition in the future (A), and that at some later point a substitution makes the maintenance condition not hold, thus making a violation (which remains thereafter) occur, leading to a violation (V) state. In addition, another substitution makes the repair condition happen at some future after the violation has occurred but before timeout occurs. The norm terminates in a state of discharge (D).

While the *failed* state (F) (represented by the dashed part of Figure 4.1), in which the timeout has occurred without the norm having realised the repair condition after a violation, is of no use when forming the logical grounding of the model, it might still be useful when establishing the substantive aspect of the norm semantics. Therefore we desire to give meaning to the failure of a norm with respect to its non-fulfilment. With the previously defined operators O and $\mathcal{O}$ we can give a meaning to the fulfilment and the failure of a norm instance in our model in Axiom 4.7:

Axiom 4.7. *Relationship between a norm instance and its non-fulfilment seen as failure.*

Given a normative model $NM = \langle Roles, Agents, s_0, Actions, Norms, Context\rangle$ and a norm $n = \langle r, f_n^A, f_n^M, f_n^D, f_n^R, timeout_n\rangle$, with α in $Agents$, r in $Roles$, such that $enacts(\alpha, r)$:

$$\vdash_{NM} O_{f_n^R \leq timeout_n}(E_\alpha[f_n^M] \preceq f_n^D \mid f_n^A)$$
and
$$\langle \pi, i, \theta\rangle \models \neg\mathcal{O}_{\theta f_n^R \leq timeout_n}(E_\alpha[\theta f_n^M] \preceq \theta f_n^D \mid \theta f_n^A)$$
$$\Rightarrow$$
$$\langle \pi, i, \theta\rangle \models failed(\ulcorner n_\alpha^\theta \urcorner)$$

Axiom 4.7 explicitly exposes the responsibility of an agent α to bring about the fulfilment of a certain norm instance by attaching such agent to the parametrisation of the subsequent *failed* predicate in case there is no discharge. In other words, if a norm instance fails, it is possible to know who is to blame. However, grounding such concept of responsibility to the logical level is not straightforward. There are many ways to formally transport this to a lower level, such as organisational structure [Grossi, 2007], institutional power [Jones and Sergot, 1996; Governatori et al., 2002], responsibility delegation [van der Torre, 2003] or directed obligations [Dignum, 1999; Carmo and Pacheco, 2000].

It is not in the scope of our work to make a choice in this respect; on the contrary: at the operational level, we will assume that the ultimate responsibility lies with the agent that is assigned to the instantiation of the norm according to its role. As an immediate consequence of this choice, our semantics will not suffer from additional complexity. Moreover, changing our semantics to accommodate a more fine-grained definition of responsibility – again, out of the scope of our work – would have less impact than having to replace any of the aforementioned options.

Therefore, at the logical level, there is no connection between: 1) *who* sees to it that the norm instance is fulfilled, and 2) *who* produces the set of actions needed to realise such fulfilment. This means that the only formal binding between an agent α and its corresponding failures – i.e. the instantiations of the $failed$ predicate – is the agent part of the tuple defining the norm instance.

Another important thing to note at this stage is that violation is not explicit in the temporal formulas. Instead, we implicitly express it in the third condition of the formula in Definition 4.6 by allowing the maintenance condition to become false at some point in time, after the norm instance has been activated. However, for monitoring mechanisms it is necessary to include the notion of violation in operational terms, in order to be able to detect such states throughout time. Also, in a norm status monitoring system, we cannot guarantee that a norm will be always fulfilled (following the non-dashed states in the lifecycle), therefore we need to make sure that the failed state is modelled too within the operational semantics.

We do this by seeing the lifecycle defined in Figure 4.1 as an transition automaton. Transition properties that define how the norm changes its status while events (world changes that modify the predicates' truthness) are occurring can be easily extracted. We are interested in directly representing these transitions in operational terms as it is useful when dealing with monitoring of norms' status. The four states *active* (A), *viol* (V), *discharged* (D), *failed* (F) are described in Definition 4.8:

Definition 4.8. Norm lifecycle predicates

(i) $\langle \pi, 0, \theta \rangle \models active(\ulcorner n_\alpha^\theta \urcorner)$ **if** $\langle \pi, 0, \theta \rangle \models f_n^A \wedge \nexists \theta' : \theta' f_n^D$
and
$\langle \pi, i, \theta \rangle \models \mathbf{X} active(\ulcorner n_\alpha^\theta \urcorner)$ **if** $\langle \pi, i, \theta \rangle \models (\mathbf{X} f_n^A \vee active(\ulcorner n_\alpha^\theta \urcorner)) \wedge \mathbf{X} \nexists \theta' : \theta' f_n^D$

(ii) $\langle \pi, i, \theta \rangle \models viol(\ulcorner n_\alpha^\theta \urcorner)$ **if** $\langle \pi, i, \theta \rangle \models active(\ulcorner n_\alpha^\theta \urcorner) \wedge \nexists \theta' : \theta' f_n^M$
and
$\langle \pi, i, \theta \rangle \models \mathbf{X} viol(\ulcorner n_\alpha^\theta \urcorner)$ **if** $\langle \pi, i, \theta \rangle \models viol(\ulcorner n_\alpha^\theta \urcorner) \wedge \mathbf{X} \nexists \theta' : \theta' f_n^R$

(iii) $\langle \pi, i, \theta \rangle \models \mathbf{X} discharged(\ulcorner n_\alpha^\theta \urcorner)$ **if** $\langle \pi, i, \theta \rangle \models active(\ulcorner n_\alpha^\theta \urcorner) \wedge \mathbf{X} \exists \theta' : \theta' f_n^D$
and
$\langle \pi, i, \theta \rangle \models \mathbf{X} discharged(\ulcorner n_\alpha^\theta \urcorner)$ **if** $\langle \pi, i, \theta \rangle \models viol(\ulcorner n_\alpha^\theta \urcorner) \wedge \mathbf{X} \exists \theta' : \theta' f_n^R$

(iv) $\langle \pi, i, \theta \rangle \models \mathbf{X} failed(\ulcorner n_\alpha^\theta \urcorner)$ **if** $\langle \pi, i, \theta \rangle \models viol(\ulcorner n_\alpha^\theta \urcorner) \wedge \nexists \theta' : \mathbf{F}_{\leq timeout_n} \theta' f_n^R)$

(i) says that the norm remains in *active* status until there is no instance of discharge condition occurring. **(ii)** says that the norm moves from the *active* to the *viol* state and remains there if there is no instance of the maintenance condition. **(iii)** says that the norm moves from the *active* to the *discharged* state if there is an instance of the discharge condition occurring and that the norm moves from the *viol* to the *discharged* state if there is an instance of the repair condition occurring. **(iv)** says that the norm moves from the *viol* to the *failed* state if timeout occurs. We make a note here, that both *discharged* and *failed* are non-persistent, meaning that if they become true, this lasts for one state in the execution path. An example showing the lifecycle timeline and the lifecycle predicates of a norm that gets violated at some point are depicted in Figure 4.2.

4.3.4 From Norm to Norm Instances

As discussed in Section 3.4.1, having formally defined instances, the compliance of a norm will be stated based on the fulfilment of each of its instantiations. We say that a norm has been complied with up to a certain time if and only if, each one of

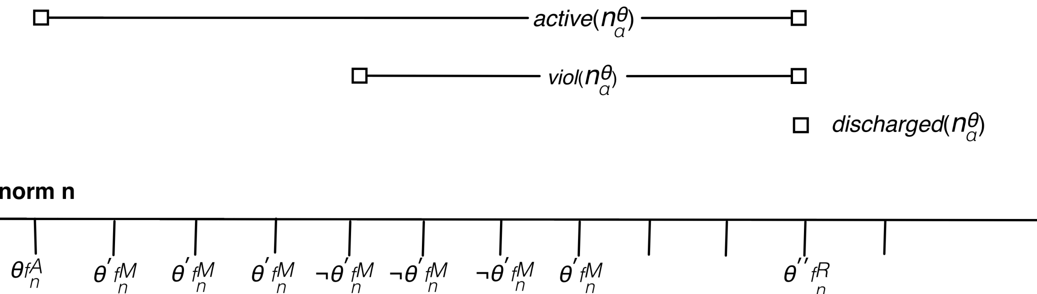


FIGURE 4.2: Norm Instance Timeline

At some point the activating condition f_n^A becomes true for some substitution θ, making the norm instance n_α^θ *active*. After this, for some substitution θ', maintenance condition f_n^M holds until some point where it becomes false. A *violation* is then raised for the norm instance. When a substitution for f_n^R makes the repair condition true, the norm becomes *discharged*, and *active* ceases to hold, completing its lifecycle. Note that while maintenance condition becomes true again at some point after the violation has occurred but the norm still remains violated. This is because after a norm violation occurs, the norm instance is considered to be violated independently of the maintenance condition and the only way for it to cease to be violated is when it gets repaired (and therefore discharged).

the instantiations triggered before this time have not been violated, where violated means that there has been $\neg f_n^M$ before f_n^D ever happening. This definition can be easily augmented to take into account more complex definitions of norm compliance, e.g. [Wang, 2010; Milosevic et al., 2002; Bou et al., 2007].

Now we have the apparatus needed to connect the fulfilment of a norm and the fulfilment of its instances, and give semantic meaning to the operator O proposed in Definition 4.4. This is done in Definition 4.9.

Definition 4.9. Fulfilment of a norm based on the fulfilment of its possible instances

$$\langle \pi, i \rangle \models O_{f_n^R \leq timeout_n}(E_\alpha[f_n^M] \preceq f_n^D \mid f_n^A)$$
iff
$$\forall \theta : \langle \pi, i, \theta \rangle \models \mathcal{O}_{\theta f_n^R \leq timeout_n}(E_\alpha[\theta f_n^M] \preceq \theta f_n^D \mid \theta f_n^A)$$

Informally: *the norm is fulfilled if, and only if, for each possible instantiation of f_n^A through time, the obligations of the norm instances activated by f_n^A are fulfilled.*

4.4 Reduction to Deontic Logics

As we have discussed, our formalisation is founded on a practical norm lifecycle and grounded onto a particular temporal logic that will allow us to implement it on already existing lightweight technologies. However, it is also important to show that there is a connection between this formalisation and some subsets of deontic logics.

The motivation behind this is to allow propositions expressed in these subsets into our formalisation by defining transparent and simple reductions. This section explores some examples of such reductions, and we discuss the formal properties of the resulting semantics.

First of all, we analyse the reduction to *achievement obligations* and *maintenance obligations*. Both are used in an indistinguishable manner in Standard Deontic Logics (SDL), the main axioms of which are:

- K: O(A→B) → (O(A) → O(B))
- D: O(A) → ¬O(¬A)[12]
- Necessitation: $\models \alpha \Rightarrow \models O(\alpha)$

In Dyadic Logic, a dyadic operator O(A|B) is used to indicate conditional obligation. Axioms, as defined in [von Wright, 1971] are:

- K1: O(A∨¬A|B)
- K2: ¬(O(A|B)∧O(¬A|B))
- K3: O(A∧A'|B) == O(A|B)∧O(A'|B)
- K4: O(A|B∨B') == O(A|B)∧O(A|B')
- K5: P(A|B) == ¬O(¬A,B)

4.4.1 Reduction to Achievement Obligations

In achievement obligations a state has to be accomplished by an agent at some point in the future: $O(A)$. There is no activation, maintenance nor reparation. Therefore, A can be seen purely as a discharge:

- $f_n^A \equiv \top$
- $f_n^M \equiv \top$
- $f_n^D \equiv$ A
- $f_n^R \equiv \bot$

In this context, our fulfilment formula is reduced, with respect to Definition 4.6, to:

Definition 4.10.

$$\langle \pi, i, \theta \rangle \models \mathcal{O}_{\theta f_n^R \leq timeout_n}(E_\alpha[\theta\top] \preceq \theta f_n^D \mid \theta\top)$$
iff
$$\langle \pi, i, \theta \rangle \models \exists j \geq i, \theta'' : \langle \pi, j, \theta\theta'' \rangle \models f_n^D$$

[12]As explained in Section 2.2.2.2.3 Von Wright defined axiom D as D: O(A) → P(A) [von Wright, 1951]. However, other logicians prefer to write this as O(A) → ¬O(¬A) given that permission P is defined as P(A)=¬O(¬A) [Hilpinen, 1971].

The detailed extraction of the substitution can be found in Appendix C.1.1. Definition 4.10 fulfils axioms *K* and *Necessitation*, but not *D*. While this result might seem counter-intuitive, there are some interesting implications that will be discussed in Section 4.7.

4.4.2 Reduction to Maintenance Obligations

In maintenance obligations a state has to be maintained by an agent at all times: $O(A)$. There is no activation, discharge nor reparation. Therefore, A can be seen purely as a maintenance condition:

- $f_n^A \equiv \top$
- $f_n^M \equiv \text{A}$
- $f_n^D \equiv G(f_n^M)$ [13]
- $f_n^R \equiv \bot$

In this context, our fulfilment formula is reduced, with respect to Definition 4.6, to:

Definition 4.11.

$$\langle \pi, i, \theta \rangle \models \mathcal{O}_{\theta f_n^R \leq timeout_n}(E_\alpha[\theta f_n^M] \preceq \theta f_n^D \mid \theta f_n^A)$$
iff
$$\langle \pi, i, \theta \rangle \models \forall \theta' : \mathbf{G}(\theta' f_n^M)$$

The detailed extraction of the substitution can be found in Appendix C.2.1. Definition 4.11 fulfils axioms *K*, *D* and *Necessitation*, so there is a valid reduction from our formalisation to maintenance obligations in the context of SDL.

A direct consequence of having a valid reduction in which the axiom *D* is fulfilled is that we can therefore safely assume that, if we define an operator $\mathcal{F}$ that represents a prohibition as in [Hilpinen, 1971], where prohibition is defined as F(A)=O(¬A):

$$\langle \pi, i, \theta \rangle \models \mathcal{O}_{\theta f_n^R \leq timeout_n}(E_\alpha[\theta f_n^M] \preceq \theta f_n^D \mid \theta f_n^A)$$
iff
$$\langle \pi, i, \theta \rangle \models \neg \mathcal{F}_{\theta f_n^R \leq timeout_n}(E_\alpha[\neg \theta f_n^M] \preceq \theta f_n^D \mid \theta f_n^A)$$

[13] Because of the nature of SDL, there is no inherent mechanism that allows to define an *end* of the norm, and therefore, we need a way in our reduction to indicate that the norm is active forever. Therefore, we use $G(f_n^M)$ as the discharge condition. It can be read as: *until the end of time, this norm instance is always being maintained*. An obvious consequence of this is that the norm will be never fulfilled – but, again, this would never happen either in a maintenance norm expressed in SDL. However, failed instances will be properly detected anyway.

4.4.3 Reduction to Dyadic Maintenance Obligations

In dyadic maintenance obligations, a state A, which the agent has to maintain at all times since the moment another state B occurs (written as $O(A|\ B)$) is declared. There is no discharge nor reparation. Thus, A can be seen purely as a maintenance condition and B as an activating condition:

- $f_n^A \equiv$B
- $f_n^M \equiv$A
- $f_n^D \equiv G(f_n^M)$ [13]
- $f_n^R \equiv \bot$

In this context, our fulfilment formula is reduced, with respect to Definition 4.6, to:

Definition 4.12.

$$\langle \pi, i, \theta \rangle \models \mathcal{O}_{\theta f_n^R \leq timeout_n}(E_\alpha[\theta f_n^M] \preceq \theta f_n^D \mid \theta f_n^A)$$
iff
$$\left[\forall j \ \geq i : \langle \pi, j, \theta \rangle \models \neg f_n^A \vee (\exists j \geq i : \langle \pi, j, \theta \rangle \models f_n^A \wedge \forall j \geq i, \theta' : \langle \pi, j, \theta'\theta \rangle \models f_n^M\right]$$
$$\wedge \forall i \leq k < j, \langle \pi, k : \theta \rangle \models \neg f_n^A$$

The detailed extraction of the substitution can be found in Appendix C.3.1. Definition 4.12 fulfils axioms $K1$, $K2$, $K3$ and $K4$ [Hilpinen, 1971], so there is a valid reduction from our formalisation to maintenance obligations in the context of Dyadic Deontic Logic.

Similarly as in Section 4.4.2, the valid axioms of this reduction allow us to define an operator $\mathcal{F}$ that represents the prohibition operator in our framework.

4.5 Example

In this section we describe an example which will be used to illustrate different aspects of our approach to normative reasoning and the utility of our framework not only throughout the rest of the chapter but also in Chapter 5.

4.5.1 Pizza Delivery Domain

The example models a pizza delivery use case. Whenever the pizza store receives a delivery order, it assigns the delivery to a specific driver-agent. From the moment the order is received, the pizza has to be delivered within a specific margin of time (which

is normally half an hour). The driver picks up the pizza whenever it is ready and delivers it. Since some time is spent in preparing the pizza, the agent normally has an even stricter deadline for its delivery, depending on how much time has been spent on the preparation. For that reason, we do not set a fixed time for the pizzas to be delivered, instead we will use varying delivery time deadlines and we will see how, combined with the norms, they influence the decision making of the agent.

The example is set on a particular existing area of Barcelona, the Eixample neighbourhood (Figure 4.3). The grid on which the agent moves comprises of streets and junctions between them. As in most traffic simulation models, we set junctions to be the primary element in the domain and model streets as connections between junctions. Throughout this section all elements are described in the textual form of the elements of Section 3.2. Predicate and action arguments are denoted with a '?' in front of them as it gives a clearer visual appearance. We will omit the definition of the low level elements (ontological and contextual elements defined by the conceptual framework in Section 3.2) as they are easily deducible from the context and the explanations. The details of the example can be found in Figures B.1-B.5 in Appendix B.1.

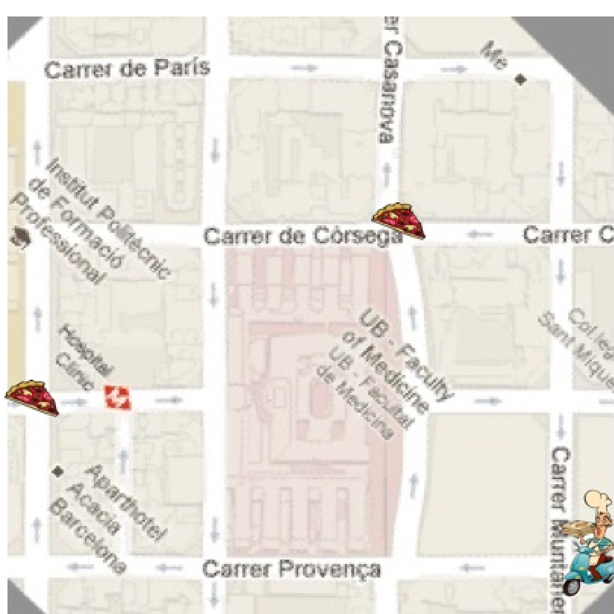


FIGURE 4.3: Example Domain

The agent is always at some junction (an intersection of two streets) within the map (**vehicle_at**(?main,?street) predicate, with main indicating the street on which the agent lastly was moving). The agent moves from one junction to another provided that there is a street connecting them (**connection** predicate). For simplicity we assume that the delivery is only done at junctions (and not in between junctions). Still, in a more advanced model it is fairly easy to include delivery points at any building on a street, provided we have a more detailed version of the neighbourhood domain. Each street might have a directional (possibly bidirectional) way for the cars to move. Each street has an associated speed limit .

One special function that we distinguish in our example is **p_time** (we refer to it as **p_time** because it is often a term reserved by several programming environments), representing time. We consider that not all time steps have equal time duration, therefore time does not increase by one for each action taken (some transition from one state to another). Instead, we give a duration flavour to each action, meaning that time increases at the end of each action execution correspondingly. In reality, the implementation of time in every case will be similar, since as we will see, time is represented by a simple function that changes its value at every action definition. We consider **p_time** to be more flexible since, e.g. in case of the existence of multiple actors, it might be a separate function for each of them. At the initial state, **p_time** is 0 and with every action taken time increases accordingly. Moving from a junction to the next requires time proportional to the speed of the vehicle. When an individual drives with `high` speed (around 60 km/h), this time is 1 min, when he drives with `medium` speed (around 40 km/h), this time is 1.5 min and when the speed is `low` (around 20 km/h), this time is 2mins. Delivering a pizza takes 3 minutes.

As in the real world, we also assume that the goal for an agent is to have the pizzas delivered at their destination. Therefore, a solution to the problem will be a plan which ends at a state where the agent has delivered all the pizzas that were to be delivered. In our example the agent must make two deliveries. One at the junction (`corcega`, `casanova`) and the other one at (`rosello`, `urgell`).

Having explained the main elements featuring in the example (when dealing from the simplistic point of view where no norms are mentioned or applied yet), the model for the actions **MoveBetweenJunctions** and **DeliverPizza** can be seen in Table 4.1. The preconditions and effects contain logical expressions consisting of predicates, functions with logical connectors such as `and/or`, logical equivalences etc., between them.

Action **MoveBetweenJunctions**`(?v, ?main, ?sideStreet1, ?sideStreet2, ?speed)` represents the transfer of vehicle `v` from a junction to another. Before the execution the vehicle is at some junction (of `sideStreet1` and `main`) and after the execution the vehicle is found at another junction (of `sideStreet2` and `main`), the two connected by a `main` street. It is assumed that in the predicate **MovedBetweenJunctions** models exactly this agent's last transition. The action might be performed with aforementioned different types of `speed` which can be `low`, `medium` and `high`. The speed in which a vehicle is currently moving is represented by the function **vehicleSpeed**. Note that the action physically allows moving either way between junctions, even if there only is a single directional connection between them.

Action **DeliverPizza**`(?p,?street1,?street2)` models the act of a person `p` delivering a pizza at a specific location, which is a junction between two streets, `street` and `street2`. It can be performed under the condition that the agent has a pizza

to deliver at this location (predicate **hasPizzaFor**) and results in the pizza being delivered (predicate **pizza_delivered**) at the location.

Action **GetOffBike**(?p,?v) models the simple act of a person p getting of the motorbike v, and thus ceasing to be driving the vehicle.

Action	Description
MoveBetweenJunctions	parameters: ?v ?main ?sideStreet1 ?sideStreet2 ?speed precondition: (vehicle_at(v,main,sideStreet1) or vehicle_at(v,sideStreet1,main)) and (connection(main,sideStreet1,sideStreet2) or connection(main,sideStreet2,sideStreet1)) effect: ¬ vehicle_at(v,main,sideStreet1) and ¬vehicle_at(v,sideStreet1,main) and vehicle_at(v,main,sideStreet2) and (movedBetweenJunctions(v,main,someStreet,sideStreet1) -> ¬movedBetweenJunctions(v,main,someStreet,sideStreet1)) and (movedBetweenJunctions,v,sideStreet1,someStreet,main) -> ¬movedBetweenJunctions(v,sideStreet1,someStreet,main)) and movedBetweenJunctions(v,main,sideStreet1,sideStreet2) and ((speed=low) -> (p_time+=3) and (vehicleSpeed(v)=20)) and ((speed=medium) -> (p_time+=2) and (vehicleSpeed(v)=40)) and ((speed=high) -> (p_time+=1) and (vehicleSpeed(v)=60))
DeliverPizza	parameters: ?p ?street1 ?street2 precondition: driving(p,v) and (vehicle_at(v,street1,street2) or vehicle_at(v,street2,street1)) and hasPizzaFor(p,street1,street2) effect: pizza_delivered(p,street1,street2) and pizza_delivered(p,street2,street1) and ¬hasPizzaFor(p,street1,street2) and ¬hasPizzaFor(p,street2,street1) and p_time+=3
GetOffBike	parameters: ?p ?v precondition: driving(p,v) effect: ¬driving(p,v) and p_time+=0.5

TABLE 4.1: Actions description, expressed in the textual form of the metamodel in Section 3.2

4.5.2 Norms

In order to add the norms in our example, we introduce two extra functions, on top of **p_time**. Firstly, the system maintains a penalty system for each individual driver **penalty_points**, or simply **points**) based on his or her ability (or inability) to deliver pizzas on time. Each agent starts with an initial amount of 0 points. Furthermore, an agent disposes of an amount of **fine** credits (it could be translated to money). This factor is important since throughout the delivery simulation, as we will see, fines might be imposed as a result of traffic violations. We assume an initial amount of 0 units.

We assume that the weight for the three parameters **p_time**, **penalty_points** and **fine** in our example varies, and throughout the experiments in this and the following chapter we will try various combinations and see how they result in different behavioural outcomes. The domain actions that model the reduction of the factors **penalty_points** and **fine** are **PayFine1** and **PayFine3** respectively, and **AugmentPoints** for the **penalty_points**, and the description for them can be found in Table 4.2.

PayFine1	parameters: p precondition: driving(p,v) effect: fine_paid1(p) and fine(p)+=30
AugmentPoints	parameters: p precondition: effect: points_augmented(p) and penalty_points(p)+=10
PayFine3	parameters: p precondition: driving(p,v) effect: fine_paid3(p) and fine(p)+=30

TABLE 4.2: Repair actions description, expressed in the textual form of the metamodel of Section 3.2

We now introduce in our example three norms (listed in Table 4.3). We assume the existence of a role *citizen* and a person called *sergio*, who enacts this role. All three norms address the agents that enact the role *citizen*. The first norm (**norm1**) concerns wrong-way driving. Normally, this is an serious offence under the traffic law. But, given that an agent might nonetheless be able to drive the wrong direction in a street, the action is still physically possible. Given that, we introduce a norm stating that whenever an agent causes such an infraction, this will result in the agent having to have a fine paid. Another norm (**norm2**) indicates that whenever a pizza has been ordered, the agent is obliged to deliver it within the specified time limit set by the pizzeria. On the contrary case, the agent will have penalty points augmented. The

norm1:Prohibited to go in opposite direction to traffic	
roles	citizen
modality	O[a]
f_n^A	driving(p, v)
f_n^M	¬movedBetweenJunctions(v, main, street1, street2) or connection(main, street1, street2)
f_n^D	¬driving(p,v)
f_n^R	fine_paid1(p) and ¬driving(p,v)
norm2: Obliged to deliver on time	
roles	citizen
modality	O
f_n^A	hasPizzaFor(p, street1, street2)
f_n^M	p_time < goalpizzatime(street1, street2)
f_n^D	pizza_delivered(p, street1, street2)
f_n^R	points_augmented(p) and pizza_delivered(p, street1, street2)
norm3: Obliged to drive at less than the maximum speed permitted for each street	
roles	citizen
modality	O
f_n^A	driving(p, v)
f_n^M	¬(vehicle_at(v, street1, street2) and (speedLimit(street1) < vehicleSpeed(v)))
f_n^D	¬driving(p,v)
f_n^R	fine_paid3(p) and ¬driving(p,v)

TABLE 4.3: Example norms, expressed in the textual form of the metamodel of Section 3.2

[a] This norm is essentially a prohibition modelled as an obligation. As explained in Section 3.2.6, prohibitions can be modelled as obligations, by negating the maintenance condition.

third norm (**norm3**) has to do with the speed of a vehicle. An agent cannot drive faster than the speed limit of the street he is in. On the contrary case, he will receive a fine to be paid. An example of how one of the norms, **norm3** is represented in terms of our conceptual framework can be found in Figure B.6 of Appendix B.1.

We assume that there are observers. These observe the environment for possible infractions. Following the real-world conditions in our case, observers could be easily introduced when monitoring the time which an agent takes to deliver a pizza, by the use of a GPS device tracking his movement. Additionally, as happens in big cities, CCTV cameras or police agents might exist at various spots within the city detecting traffic infringements.

4.6 Planning with Norms

Although Definition 4.8 is sufficiently expressive while implementing a monitoring framework, it cannot be applied in a planning system. This is because most planners allow modelling the transitions between states (actions) in a way such that there is exclusive dependency on the values of the previous state's properties. In this way, for example the *active*() status of a norm cannot be easily expressed, since not only does it need to be aware of the activeness at the previous state, but it also needs to be aware of whether at the current state the discharge condition occurs. The use of extra predicates such as *previous*() and *current*() that provide such functionality, allowing to check whether formulas hold in current and previous states, is permitted in some planning frameworks such as TLPLAN [Bacchus and Kabanza, 2000] but it proves to be costly when extensively used[14].

An alternative, on which we base our implementation, is to use Definition 4.6 and therefore languages that support the use of LTL formulas to restrict the plans produced. Such an attempt is PDDL 3.0 [Gerevini and Long, 2006]. PDDL 3.0 specification extends PDDL [Fox and Long, 2009] with strong and soft constraints (expressed in LTL formulas) which are imposed on plan trajectories, as well as strong and soft problem goals, which are imposed on a plan. Nevertheless, it appears to be insufficient when trying to capture the semantics that we use for the norm lifecycle mainly due to two reasons:

1. it lacks the operator "until", which would permit us to express the norm lifecycle (e.g. a norm is violated when activated at some point, the maintenance condition does not hold at some state after this, and, the discharge condition does not hold at any state in between) and
2. a norm can be activated and discharged (and possibly violated) several times during the execution of a plan, something not possible to be expressed in PDDL 3.0.

TLPLAN [Bacchus and Kabanza, 2000] on the other hand, applies LTL formulas (called *control rules*) to a forward chaining search, reducing in this way the search path by pruning paths that do not comply with the rules. TLPLAN is based on (STRIPS-like) semantics that can be easily reduced to PDDL. Furthermore, TLPLAN supports operators such as $\mathbf{F}_{\leq}$ (called '*T-Eventually*' in TLPLAN), which we used when we defined the norm lifecycle. We chose TLPLAN as it contains a complete, robust and rather fast implementation, also allowing extra features such as existential and universal quantifiers. We explain below how the norms are introduced into the planning mechanism.

[14] We performed some tests on several planner implementations where we checked if a formula held in the current or previous states, and the results in terms of execution time were far worse than the ones that we present in Section 4.6.6.

4.6.1 Plans and Actions

In classical planning it is assumed that the agent executing the plan is the only source of change in the environment. As summarised in Section 3.2.5 *actions* specify what the agents are able to perform and map worlds to new worlds (states). We use an extended version of the standard STRIPS [Fikes and Nilsson, 1972] representation of the worlds, as used in PDDL [Ghallab et al., 1998; Fox and Long, 2009] and ADL [Pednault, 1994], which is considered to be the continuation of STRIPS, offering advanced semantics for the representation of actions. In this representation each state is represented as a complete list of the ground atomic formulas that hold. The closed world assumption is employed, so that every ground atomic formula not in a states's database is falsified.

Having language $\mathcal{L}$, action α is defined as[15] $\alpha =< C_{prec}, C_{effect} >$ and C_{prec}, C_{effect} are sets of fluents or negated fluents from $\mathcal{L}$. For two states s_{i-1} and s_i we have a transition from s_{i-1} to s_i through α and write it as $s_i = \alpha(s_{i-1})$ when $s_i = s_{i-1} \cup \{h|h \in C_{effect}\} \setminus \{l|\neg l \in C_{effect}\}$.

A *plan*, which we take to be a finite sequence of actions[16], generates a finite sequence of worlds (a *path* or *trajectory*). That is, a plan $\pi = [\alpha_1, \alpha_2, \ldots, \alpha_m]$ generates a trajectory $\tau =< s_0, s_1, \ldots, s_m >$ such that s_0 is the initial world and $s_i = \alpha(s_{i-1})$.

4.6.2 Types, Numeric Expressions, Conditions and Effects

ADL [Pednault, 1994] (and as a consequence most planners that implement similar semantics) offers the possibility to express a *type structure* for the objects of a domain, with a basic type `Object` being the default. In this way, actions can constrain their arguments to specific types. An example of a definition of a type `person` and some person *john* of that type (the type denoted by the '-' symbol after the object) can be:

```
(:objects
    john - person)
```

Numeric expressions [Fox and Long, 2009] (which allow for plan metrics) are formed by using arithmetic operators to combine primitive numeric expressions. Primitive numeric expressions are terms formed by applying domain *functions* to domain objects. Functions are defined by their name and argument types and they can only be of type $\texttt{Object}^n \rightarrow \mathbb{R}$, where n is the (finite) function arity. An example of a function representing the amount of money a person possesses can be:

```
(money ?p - person)
```

[15] The action definition comes in accordance with the conceptual framework definitions given in Section 3.2.5.

[16] The plan definition comes in accordance with the conceptual framework definitions given in Section 3.2.7.

ADL is expressed through a prefix syntax for all predicates. The arguments to predicates or values of action parameters cannot include numeric expressions. Numeric expressions can only appear in comparisons between pairs within a precondition. Primitive numeric expressions might get updated within effects. This is done by assignment operations such as *increase* and *decrease*. Values must all be built from primitive numeric expressions defined within a domain and manipulated by the actions of the domain. They can represent quantities of resources, accumulating utility, indices or counters. An example of an effect condition increasing the money the individual 'john' possesses by 100 could be:

```
(increase (money john) 100)
```

It has to be noted that according to each representation's language specification (for example PDDL 2.1 [Fox and Long, 2009] or TLPLAN [Bacchus and Kabanza, 2000]), the syntax of types, functions, numeric expressions, conditions and effects can slightly vary.

4.6.3 Calculation of Plan Cost

In ADL-derived planning domains (such as TLPLAN [Bacchus and Kabanza, 2000] or PDDL 2.1 [Fox and Long, 2009] domains), actions can have an associated cost function which might be combined with conditional effects. This allows for paths (plans) to have a total cost. We include action costs in our framework in order to be able to evaluate not only the norm conformance but also to calculate the most beneficial paths while taking into account compliance with or violation and reparation of norms. As mentioned in the previous sections, when defining what the norm repair (penalty) state is, we require that any plan solution that violates a norm also takes the necessary steps to achieve its repair state. Again, by giving the appropriate cost function to the actions, the planner is able to determine the choice between a path that complies with a norm and a path that violates it and afterwards compensates by reaching the repair state. In this way, such complex reasoning over conformance to the soft constraints imposed by norms becomes part of the planning problem rather than an external issue.

4.6.4 The Normative Planning Problem

We are now able to define the exact nature of what normative planning is. In Definition 4.9 we gave the notion of a norm's fulfilment with respect to the fulfilment of its instances. We will use and apply this definition in order to formalise the problem of means-ends reasoning with the influence of norms.

In our framework, normative planning is formalised as a planning problem in a domain where there are norms which acquire committing (obligation) force through

planning paths. Additionally, when the agent fails to comply with one, then he needs to see to it that it gets repaired. Having that in mind, the problem then is to find such a plan that the final state achieves the goal(s)[17] and that there are no pending norms[18] (they have all completed their norm lifecycle). We wish to compute only plans where norms have fully completed their lifecycle since, from the moment obligations get into force, they cannot indefinitely exist without being accomplished, as this would remove the deontic weight that each norm holds[19]. In this way we force all norms that are activated through the different trajectories to be either fulfilled or repaired to make the agent conscious about the effects of each norm violation. In short, the planning problem is defined as finding a plan such that:

- the final state achieves the goal,
- there are no pending norms.

Formally, given a normative model $NM = \langle Roles, Agents, s_0, Actions, Norms, Context \rangle$, a goal g (which can be a set of goals) we are looking for a plan $\pi = [\alpha_1, \alpha_2, \ldots, \alpha_m]$ generating the trajectory $\tau =< s_0, s_1, \ldots, s_m >$ where all goals are accomplished and additionally the norms have followed the lifecycle:

- $\langle \pi, m, \emptyset \rangle \models g$
- For all norms $n = \langle r, f_n^A, f_n^M, f_n^D, f_n^R, timeout_n \rangle$ in $Norms$ and all agents α in $Agents$ and roles in $Roles$ such that $enacts(\alpha, r)$, we have that:

  $\langle \pi, 0 \rangle \models O_{f_n^R \leq timeout_n}(E_\alpha[f_n^M] \preceq f_n^D \mid f_n^A)$

While there might be multiple plans that satisfy the above conditions, we are additionally interested in the "preciousness" of them. By default the agent sets as criteria for the preciousness of a plan the minimisation of the addition of all the numerical factors. A more complex formula comprising of weights of importance for every factor might be defined within the agent and passed on to the normative planner[20].

The costs of a plan can be calculated in different ways. One can take the costs of repair actions completely into account, but also halve them when, e.g. the chance that someone discovers a violation is 0.5 and thus half the times repair actions do not have to be executed. This of course depends on the agents own morals as the detection

[17]We could investigate a more general case where some of the goals are just objectives to be completed not at the end of the plan, but at any intermediate state. Current planners such as TLPLAN can offer this possibility through formulas of the type **F**g, indicating that a goal should be satisfied at some point during the execution path

[18]By the term "pending" we refer to non-fulfilled norms, with fulfilment being represented by the operator O.

[19]It could indeed be the case that the goal state is reached before an obligation is accomplished. In that case, the plan should make sure to reach the goal at some state and then "close" the pending obligations by taking the necessary steps. That would require the computed path to have the goal state achieved at some step and the norms having completed their lifecycle and becoming non-active by the final state. This alternative representation of the planning problem can be created by including the reaching of the goal, Fg, as part of the temporal formulas instead of forcing it to be the end state.

[20]In the rest of the document we will use the term *normative planner* to refer to a planner (whether this is TLPLAN, used in this chapter or, Metric-FF, used in Chapter 5) running such a problem.

chance might be of less interest to some agents. In addition, one can see the exercise of creating a plan and choosing the one with lowest cost as just one way of choosing the next action. It is important to note here that this does not mean that the plan will always be executed completely. What happens in this case is to choose a next action BECAUSE it is the first one of a possible plan with the lowest cost to achieve a goal. But in principle the plan could be recomputed after each action, before a next action is chosen. Of course the overhead for replanning would have to be taken into account in the process but this is out of the scope of the thesis.

A formula "**fun**" that represents the agent's total interests, incorporating the different weights for each possible factor can be used to reflect the preciousness of a plan. Such a formula can be expressed through our normative framework described in Section 3.2 as a function element of our metamodel. We reserve the special name **fun** to indicate the specialness of this function so that the agent can process, translate and insert it to the planning problem. In general **fun** will need to be defined by the designer together with the rest of the metamodel, and carefully make sure that it reflects appropriately and in a balanced way the factors that the agent wants to be taken into account. Consequently, **fun** can be seen as the reflection of what the agent considers to be the best optimisation setting for its various factors (and can be considered as its "character", since different weights will indicate different characteristics that he might have, for example substantial weight put on a *time* factor could indicate an agent that desires to be punctual).

4.6.5 Implementation via a Modified TLPLAN Algorithm

In order to create a normative planner, we represent the norms through Definition 4.6 as control rules within the planning domain. This implies that, for every norm, we need to create such a control rule. The conjunction of all those rules will be the final control rule. Figure 4.4 contains the part of the control rule for **norm1**. Figure 4.4 also depicts our implementation of the normative planner. The problem file remains intact, while for the set of norms, the control rule is created and added to the domain file.

With the use of the control formula, during the execution, the planner will only allow paths where a norm never gets instantiated, or where a norm gets instantiated and never violated or where a norm gets violated but repaired before the specified timeout is reached. Therefore, the planner will never allow for a plan that includes a norm instantiation to be violated and never get repaired to be produced. That is, it discards the ones that do not conform to the norm lifecycle. The system allows for multiple instantiations to be checked throughout the execution paths.

The TLPLAN algorithm [Bacchus and Kabanza, 2000] works by forward searching through the state space, *progressing* the control formula through the sequence of states generated during planning search. However, the progression algorithm executed by TLPLAN is not complete. This is because while it might detect some falsification of the

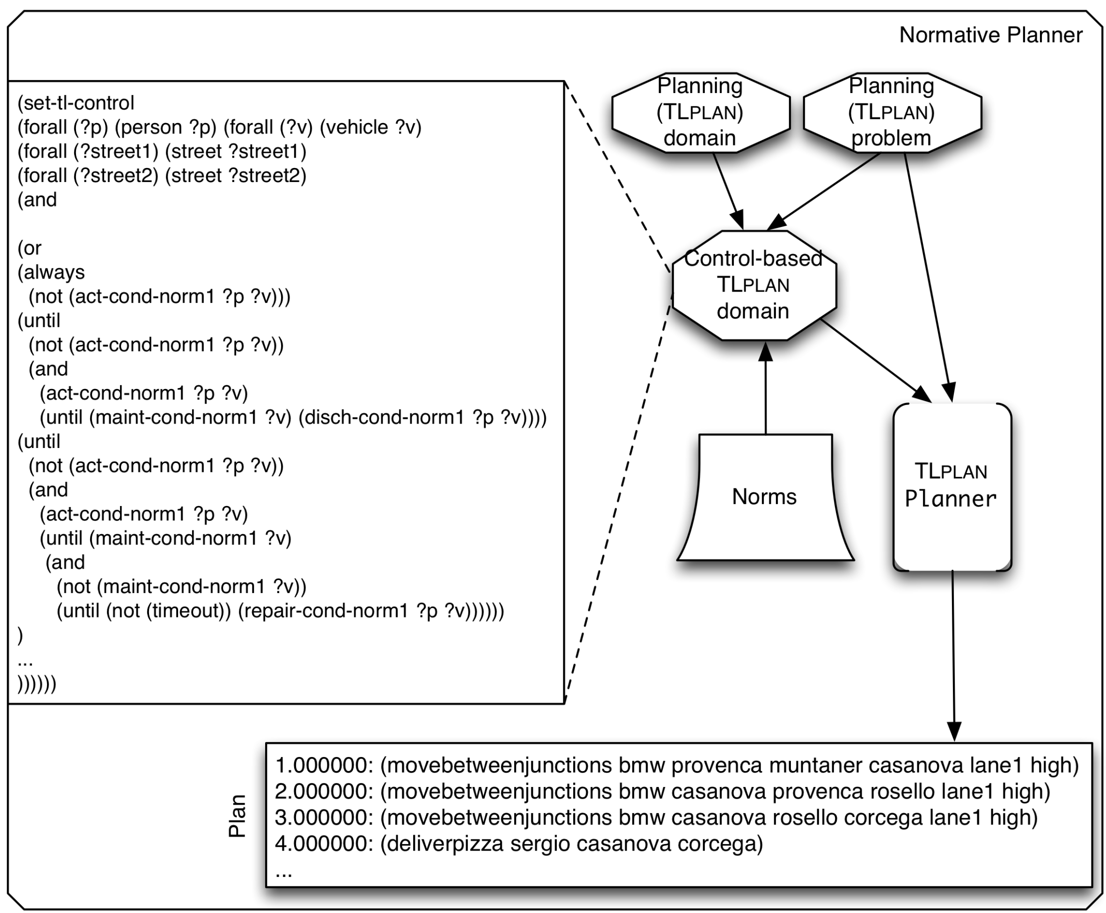


FIGURE 4.4: Normative Planner using TLPLAN

control formula at some state (thus excluding branches from the search state), it cannot reason about all possible sequences that "could" be generated (this would require a theorem prover). Instead, it can only check if the control formula is falsified by the sequence of states generated so far. Thus, progression often has the ability to give us an early answer to our question, but it cannot always give us a definite answer. Given just the current world it does not have sufficient information to determine whether or not these eventualities will be achieved in the future. In other words, eventualities are essentially useless for search control as performed by TLPLAN.

We performed some changes to the TLPLAN code, as we needed the planner to take a closed world assumption in the temporal model and behave as if the temporal path is finite, that is, that the time of the world ends at the point the plan ends. From a practical point of view, the main change was done in the search algorithms. Whenever a plan was found as being compliant with the planning goal, a final check was done in which the temporal control formula was progressed to the end of time, that is, as if the current plan ending time was the infinite. A parallel implementation for the progression formula was therefore made which is only executed at each point where

the planner has found a goal-fulfilling plan, with the following behaviour. At a state reaching the goal(s):

- If a formula is $Eventually(\psi)$, or $T\text{-}Eventually(t,\psi)$, or $Until(\phi,\psi)$, it progresses to $\bot$. If one or both arrive at the end of the plan, that means that ψ has never been true.
- If a formula is $Always(\psi)$, it progresses to $\top$. If an $Always(\psi)$ arrives at the end of the plan, that means that α has always been true.
- The rest of the formulas progress as usual.

Because we force the temporal formulas to $\top$ or $\bot$, the progression of the whole temporal control formula will have a forced evaluation to either $\top$ or $\bot$. Our modification therefore allows only for goal states where the progression formula has reached a true state. Those states represent states where the goals have been achieved and at the same time the norms' lifecycle complied with our model.

TLPLAN additionally takes a formula (that is, the function **fun** mentioned in Section 4.6.4, defined in a way to reflect the agent's view over what the plans' preciousness consists of) as an input and uses it to determine the best plan that optimises the formula. We can in this way assign values to the actions that bring about the different norm conditions. Consequently the planner will be able to decide and pick between alternative plans that conform to the norm lifecycle (e.g. one that never violates and another that violates and repairs an instance of a norm) while additionally bringing the most profitable outcome for the agent.

4.6.6 Experimental Results

In this section we present our experimental results, based on the pizza delivery example[21]. As explained in Section 4.5 the three factors that count for an agent are **p_time**, **penalty_points** and **fine**. While an agent needs to spend the minimum time delivering the pizzas, at the same time it needs to maximise its points and the money it possesses. We assume that agent *sergio* starts with:

p_time = 0
penalty_points(*sergio*) = 0
fine(*sergio*) = 10

We use the symbols γ_{p_time}, $\gamma_{penalty_points}$, γ_{fine} to symbolise the different weights that we will be using for different experimental settings. Those three play a decisive role in the path chosen by the normative planner as they represent the "character" of the agent.

A naive approach would be to try to minimise the formula:

[21] The complete code for the example can be found in Appendix B.2, in Figures B.7 and B.8

fun = γ_{p_time} * **p_time** + $\gamma_{penalty_points}$ * **penalty_points**(*sergio*) + γ_{fine} * **fine**(*sergio*)

Still, given that **p_time**, **penalty_points** and **fine** have different range of values, normalisation should take place. We use standard data normalisation with **p_time** being in the range of [0..30] (the upper bound for time is an assumption), **fine** being in the range of [0..60] (30 for violation of each of **norm1** and **norm3**), **penalty_points** being in the range of [0..10] (10 will be when a violation of **norm2** occurs), and with the normalised target scale for all three factors being [0..1]. As we wish each factor to be taken into account even when having its minimum value (0), we add a constant (1) to each of their normalised values. Therefore the formula that we will feed into the planner to try minimising will be:

$$\textbf{fun}=\gamma_{p_time} * [1+\frac{\texttt{p_time}}{30}] + \gamma_{penalty_points} * (1+\frac{\texttt{penalty_points}(sergio)}{10} + \gamma_{fine} * (1+\frac{\texttt{fine}(sergio)}{50})$$

The code for the formula is depicted in Figure 4.5.

```
(def-defined-function (fun)
        (:= fun (+ (+ (* 1 (+ 1 (/ (p_time) 30)))
                      (* 1 (+ 1 (/ (penalty_points sergio) 10))))
                      (* 1 (+ 1 (/ (fine sergio) 60))))))
```

FIGURE 4.5: Metric formula in TLPLAN

With the two deliveries to be made, four possible scenarios occur, seen respectively in Figures 4.6, 4.7, 4.8 and 4.9. In the scenarios, the compliance with or violation of **norm1** is visible (the second and third norm's compliance scenarios are not simple to depict in a graphical way since they involve time and the vehicle speed):

- The first scenario (route 1) shows how the agent will first deliver the pizza at the `(casanova, corcega)` junction and then at `(urgell, rosello)`, while making a slightly longer route in order to avoid doing wrong-way driving and thus violating **norm1**.
- The second scenario (route 2) shows how the agent will first deliver the pizza at `(urgell, rosello)` and then at `(casanova, corcega)`. This implies that the route is far longer. It is the obvious choice when a fast delivery at `(urgell, rosello)` is pending while there is plenty of time for the delivery at `(casanova, corcega)`.
- The third scenario (route 3) shows how the agent will first deliver the pizza at `(urgell, rosello)` and then at `(casanova, corcega)`, but for the second one the agent chooses to go wrong way in order to arrive faster. It is a choice when a fast delivery at `(urgell, rosello)` is pending while there is not too much time for the delivery at `(casanova, corcega)`.
- The fourth scenario (route 4) shows how the agent will first deliver the pizza at `(casanova, corcega)` and then at `(urgell, rosello)`, entering a street the wrong way in order to arrive faster for the second delivery.

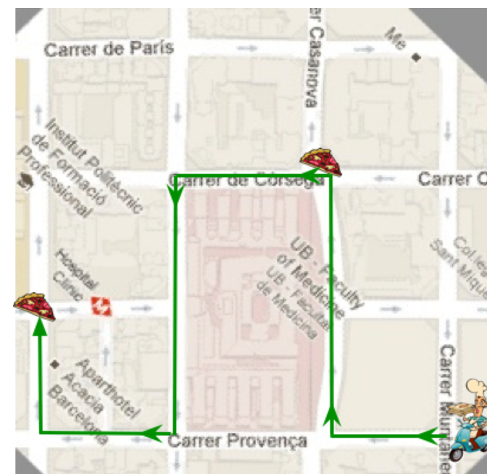


FIGURE 4.6: Route Solution 1

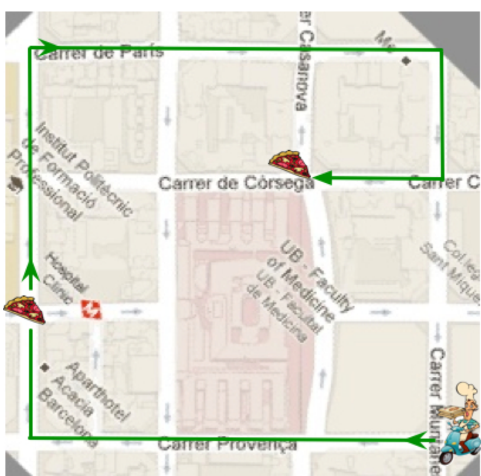


FIGURE 4.7: Route Solution 2

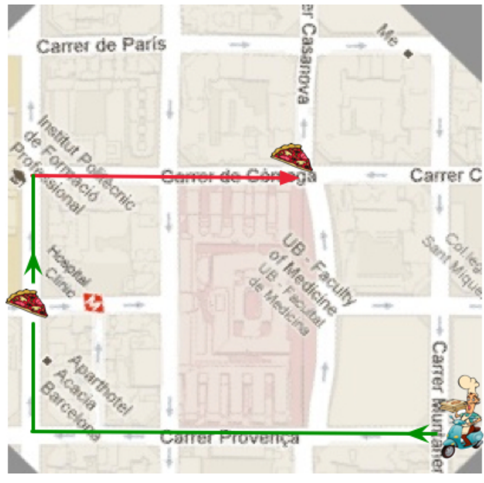


FIGURE 4.8: Route Solution 3

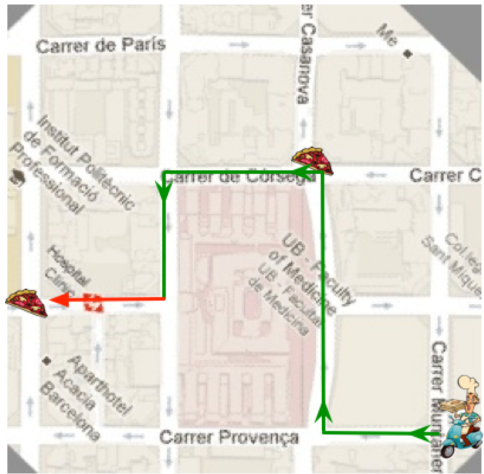


FIGURE 4.9: Route Solution 4

We have executed several experiments with our modified version of TLPLAN [Bacchus and Kabanza, 2000]. Executions were run on a Macbook Pro, running OSX 10.8.2, processor of 2.9GHz Intel Core i7, and 8GB RAM memory. The execution results can be seen in Table 4.4:

- When there is just enough time for both deliveries to be made (experiment 1), the agent will choose to make the route 1 in Figure 4.6, without having to violate any norm.
- When there is just enough time for the delivery at (`urgell`,`rosello`) to be made and enough time for the delivery at (`casanova`, `corcega`) (experiment 2), the agent will take the long route 2, delivering first at (`urgell`, `rosello`) and after at (`casanova`, `corcega`), as in Figure 4.7. But if time is slightly more pressing for the delivery at (`casanova`,`corcega`) (experiment 3), the agent will choose to take the shorter route 3, entering the wrong way in the street `corcega`.
- When time is pressing for both deliveries and it is obvious that the agent cannot make it without either violation of **norm1** and **norm3**, given the condition that he

id	fl_{p_time}	$fl_{penalty_points}$	fl_{fine}	t_{del_1} [a]	t_{del_2} [b]	route	v_1 [c]	v_2 [d]	v_3 [e]	cost
1	1	1	1	35	29	1	-	-	-	3.616
2	1	1	1	21	9	2	-	-	-	3.766
3	1	1	1	17	9	3	x	-	-	4.05
4	1	1	3	8	14.5	1	-	x	-	6.616
5	1	1	1	8	10	4	x	-	x	4.416
6	8	1	2	10	7	1	-	x	x	16.866
7	13	1	2	10	7	4	x	x	x	24.416

TABLE 4.4: Execution Results

[a] maximum delivery time for pizza delivery at `(casanova,corcega)`
[b] maximum delivery time for pizza delivery at `(urgell,rosello)`
[c] violation of **norm1** (no wrong-way driving)
[d] violation of **norm2** (deliver pizzas on time)
[e] violation of **norm3** (comply with speed limits)

really desires to avoid fines (experiment 4) he might simply choose to not deliver on time, resulting in him deliberately violating **norm2**. In a similar pressuring situation (experiment 5) but where he values equally all factors, then he might choose to take the route 4 in Figure 4.9.

- When time is pressing for both deliveries (experiments 6 and 7), making it obvious that the agent will not arrive on time for both, the amount of importance that the agent gives to time against the fine is crucial. If the agent really values his time, then he chooses not only to speed up, violating **norm3** (experiment 6), by taking route 1 (Figure 4.6), but also to short cut, by taking route 4 (Figure 4.9) and driving the wrong way (experiment 7).

The main drawback of the approach is that it is highly time consuming. TLPLAN gives the option to choose the maximum number of states searched before it fails to find a solution. The executions were performed without the use of heuristics[22] and most of them required the search limit to be set to over 100.000 states and in some cases even 1.000.000 states in order to return valid results. This resulted in execution times up to 5 minutes (and in some cases did not get to terminate), forcing us to create several files with subsets of the control rule, run them individually and then manually compare the results to get the most profitable plan. This serious disadvantage is dealt with in Chapter 5, where we modify the norm semantics and provide an advanced implementation which overcomes the barrier of time.

[22] Although the design of proper heuristics could potentially improve the efficiency of the search algorithm, we were not interested in the introduction of such heuristics, as these are domain-specific and therefore we would need either a human designer to carefully set up these heuristic functions for every domain, or ensure all norm-aware agents are capable of creating (by some automatic method) these domain heuristics from a norm specification.

4.7 Discussion

This chapter has presented work on the formal specification and development of a framework that enables modelling a deontic formalisation through temporal logic, in order to apply it in an agent's means-ends reasoning. The framework tackles at the same time three important problems related to the practical materialisation of norm-aware systems: a clear connection between the deontic level and the operational semantics, the formalisation of explicit norm instances, and the unambiguity of semantic interpretation across implementation domains. We have done so by building, based on and using the elements of the metamodel of the conceptual framework presented in Chapter 3, a connection between deontic statements and temporal logics, and between temporal logics and planning rules. Our work shows that from the latter representations the translation to the implementation level is also clear.

4.7.1 Contributions and Extensions

A contribution of this chapter has been the modification of the TLPLAN progression algorithm. While this does not completely solve the problem of fully identifying the achievement of eventualities in a formula, it manages to get round it with the reasonable (for our purposes) assumption that paths are finite and that it is of no interest what happens after the final state of a plan. Unfortunately we are not aware of any other planner implementations that successfully apply LTL control rules over paths.

Until recently, the trend when creating reasoning methodologies over norms (and even more, in works that are based on the BDI model) had been to use precomputed plans or planning rules that act as templates, which, activated by some condition, produce plans for execution [Meneguzzi and Luck, 2008, 2009b; Oren and Meneguzzi, 2013]. We view this approach as slightly disadvantaged since it does not allow enough flexibility as precomputed plans can be rigid and, in some cases, even unfeasible. Besides, in the last years, it is becoming more obvious that, independently of the norm-aware aspect of the agent's reasoning, practical agent frameworks need more advanced methods for handling their course of actions such as planning mechanisms and many of the (same) researchers have started looking into this direction [Oren et al., 2011; Oh et al., 2011]. For these reasons we find the use of a runtime planner taking into consideration the agent's current state of affairs, capabilities and objectives not only justifiable but also necessary and consider this choice of ours to use planning adapted to our normative problem's needs innovative.

A great deal of the research work that has similar aims to ours focuses on assigning labels to transition systems (either to states or to transitions) [Kollingbaum, 2005; Craven and Sergot, 2008; Hindriks and van Riemsdijk, 2013]. Although many of these approaches offer an interesting way of defining and even deciding what execution paths to avoid or exclude, in most cases it is not until an agent has already reached

the stage of picking the next action that such a restriction will be apparent. For this reason, they have somewhat weak potential to foresee distant future norm deviations and therefore lack the ability to provide an in advance complete solution towards reaching the agent's goals. On the other hand, we design our semantics and implementation in a way that violations far ahead can be expected, resulting in an informed choice of course of action since the beginning, taking the responsibility to repair where necessary.

Recent work of Alechina et al. [Alechina et al., 2013] is closely related to ours as far as the notions of norm compliance or violation and repair are concerned, but also in the use of temporal formulas over execution paths to capture norm compliance. As they also did in [Alechina et al., 2012], they apply a *conditional norm* formalisation consisting of conditions similar to our activating, discharge and repair condition, while the deontic statement is a state to be achieved (or avoided respectively in case of prohibition) at some point before the discharge and, on the contrary case, a violation sanction to be carried off. The authors proceed to create a model checking system that implements *normative update* of norms. Having a transition system, norms are verified with respect to a path expressed in CTL and a given *multiset* representing the allowed number of sanction occurrences. Being an interesting advance towards normative validation, the approach nevertheless compared to ours, has several main disadvantages: 1) It does not perform any kind of reasoning, which would allow the agent to pick or even better construct profitable execution paths. Instead, it mainly focuses on the verification aspect of norm compliance. 2) While an interesting and flexible representation of norms is used, only a brief mention of instances is made, and no real weight is given on how these might be applied on a practical level, and how norm verification can be made over several of them.

The proposal presented in this chapter might be extended in several respects. As opposed to the definitions in Section 4.3, a different representation of norm conditions could be dealing with events (actions) performed. That would mean that instead of having predicates (i.e. facts) defining the current state of affairs, the activating, discharge, maintenance and repair conditions would include actions performed (for example a discharge condition being **`DeliverPizza`**`(corcega,paris)`). Nevertheless, in order to do so, we need to establish a proper grounding with a semantics of events, such as event calculus.

In our thesis, we follow the traditional way of handling time, in order to reduce the complexity of our framework, assuming that time corresponds to time steps in a plan. However, it is not always realistic to do so. An alternative but more complicated way of modelling time would be to implement a clock, keeping track of time in the environment. Individually, each agent will have its own representation of time or a timer, which could be different to the one of the environment. JACK agent platform [Winikoff, 2005] is an example of a framework that contains three different types of clocks. In the case of clock, our framework semantics (in particular the formulas and state descriptions) need to be adjusted to include and handle the real time element. In

[Aştefănoaei et al., 2010] the authors make an attempt to model and give semantics to time when dealing with multi-agent systems and they present some interesting ideas that could be added to our framework.

Furthermore, we do recognise that the constraints on the expressiveness of the norm lifecycle automaton from Figure 4.1 are quite limiting. Different formalisms could allow us to work with a version of the lifecycle closer to the one depicted in Figure 3.33, probably in a logic framework different to the chosen LTL one. In order to tackle the restrictions that occur from the expressiveness of LTL, as well as the performance issues that occur when planning with it, in the next chapter we propose an alternative approach, based on an extended version of the norm formalism.

4.7.2 Revisiting Requirements

Table 4.5 summarises the requirements covered by the framework presented in this chapter. As we can see, the normative reasoner, can only perform a one-time deliberation, and since we have not had it yet integrated into an agent framework, the environment change (R1.5) cannot be taken into account. For the same reason, tool support (R4.5) is not yet provided. Also, the relatively slow execution time results indicate that R4.7 and R4.8 cannot be guaranteed.

Req.	Description	Status	Justification
R1.1	Deliberative, means-end, norm-oriented reasoning mechanism	✓	We apply a planning mechanism that receives norms in the form of path control rules.
R1.2	Decision making process guided by user preferences	✓	The planning mechanism supports action costs and tries to maximise the overall value of the plan, according to the criteria set by the agent.
R1.3	Goal driven decision making	✓	The planner performs its planning algorithm trying to achieve some goals. These goals are the agent's goals.
R1.4	Agent capabilities specification accommodated by framework	✓	The agent capabilities can be seen as the possible actions to be performed. The action language used supports action descriptions.
R1.5	Adjust in case of relevant environment change	✗	This feature is not covered, as in this approach we just described a one-time normative planning execution. However, if accommodated within an agent framework, it is possible to have a mechanism that adjusts to the environment change.
R1.6	Norm conflict toleration	✓	The formalisation used takes into account norm conflicts. The normative planning problem definition comes up with a solution possibly containing norm violations, but that is most profitable for the agent according to the criteria set.
R2.1	Full domain/environment definition	✓✗	The domain representation is based on ADL, a language that permits the representation of actions and domain change. However, in this chapter we have made no distinction between the agent's representation of the domain and the environment, as the framework is not integrated within an agent environment yet.
R3.1	A well defined normative model allowing the clear and unambiguous interpretation of norms on a operational level	✓	The formalism presented in this chapter provides a solid normative model and the definitions of norm compliance and of the normative problem provide a functional interpretation of norms.

R3.2	Mechanisms for agent behaviour monitoring of norms	✓✗	This mechanism has not been implemented but the formalism is created in a way that monitoring can happen. More on a monitoring mechanism based on the same semantics can be found at [Álvarez-Napagao et al., 2010; Aldewereld et al., 2010].
R4.1	Agent-oriented architecture	✓	Our formalism addresses and can be integrated within an agent-oriented architecture, where agent capabilities and preferences are seen as and translated to domain actions. The resulting plan will serve as a plan to be executed by the agent.
R4.2	Open standards support	✓	We provide a generic mechanism to support normative reasoning. We represent our normative model on the Eclipse Modelling Framework (EMF) which is openly accessible. Furthermore, the actions's representation in TLPLAN is ADL-based, a widely used standard in the planning community.
R4.3	System platform-independent model	✓	We provide a generic mechanism to support normative reasoning. We represent our normative model on the Eclipse Modelling Framework (EMF) which enables various transformations and thus interoperability with other frameworks.
R4.4	Strong focus on semantics at domain, agent context, ontology, normative level	✓	The formalisation (Normative Model, norm fulfilment, norm instances) provides a clear and functional understanding of all these elements.
R4.5	Tool Support for norm and domain representation	✓✗	A tool for norms representation is provided, since the metamodel is defined in Eclipse Modelling Framework (EMF). However, no tool is provided for the domain representation and the designer needs to express the TLPLAN domain knowledge by hand.
R4.6	Support for multiple standards and extensibility	✓	We provide a generic mechanism to support normative reasoning. We represent our normative model on the Eclipse Modelling Framework (EMF) which enables various transformations and thus interoperability with other frameworks.
R4.7	Soft real time performance	✗	As we have seen in Section 4.6.6, the results can get costly. For this reason, we do not see this requirement as satisfied.
R4.8	Reasoner's response time priority over optimality	✗	Our experiments with TLPLAN were performed without the use of heuristics. TLPLAN provides the option to use a maximum number of worlds (states) explored before it terminates the execution. However, this does not fulfil the time priority over optimality, given that in most cases setting a low limit of worlds returns no result.
R4.9	Validity	✓	In our case, validity translates to the the planner providing a correct solution to the planning problem specified [Ghallab et al., 2004]. Given that the TLPLAN algorithm ensures the correctness of the plan returned, this requirement is fulfilled.

TABLE 4.5: Requirements Analysis

The requirements not complied with and especially the execution time limitations, can serve as a motivation to the next chapter's proposal, where a faster mechanism is developed and integrated within an agent environment, providing the appropriate tools for the agent domain representation.

Chapter 5

Practical Normative Reasoning with Repair Norms and Integration into a BDI Agent

In Chapter 4 a proposal for expressing the norm's lifecycle by means of temporal rules that can impose restrictions on a planning domain was presented. The experimental results, promising as they might be, have shown that it is not always feasible to have reasonable execution time.

Additionally, while the norm representation (Section 4.3.1) and its lifecycle (Section 4.3.3) set a good semantic grounding, they are bounded by some issues:

- A specific instance can only be created once, since the lifecycle permits the activating condition and discharge condition to occur once each.
- The fulfilment of a norm is considered to have occurred whenever the instances do not get activated or their discharge condition (or repair in case of violation) occurs. However, on a practical level, setting as an objective to have instances fulfilled in this way might be counter-intuitive, as in real life, norms that possibly never reach a discharge point (a point where one is not bound anymore to them) exist. Therefore, we would like to be able to somehow include norms in a practical framework and allow them to influence the decision making procedure, without leading to dead-ends (we recall that in Section 4.3 an activated norm instance will need to get discharged at some point within the execution path that leads to the goal).
- A repair condition does not always seem sufficient to be able to handle the violation of a norm. There exist real-life scenarios where some norm is violated and a new norm comes into force from the violation of the previous one, and, if the latter is violated, an even newer norm comes into force, etc. Such situations lead to multiple layers of norms and violation handling norms, which, using the representation of Section 4.3 are impossible to model.

In this chapter we aim to tackle the above issues. More precisely:

- We modify and include into our model more complex norm representations (which extend the Chapter 4 representations) that contain primary norms as

repair norms (that might also contain their own repair norms), an idea briefly explained back in Section 3.4.3. The new formalisation also allows for multiple norm violations to occur throughout the lifecycle of a norm instance, giving the model more flexibility.

- We redefine the normative planning problem according to the new formalism. This time we use a different planner, Metric-FF and translate the problem to a PDDL domain and problem, to be solved by the planner. The new approach results in faster execution times.
- We then integrate our new normative planner within an existing BDI agent framework, 2APL. In order to do so, we need to provide a correspondence from and to the normative elements (norms, actions, plans) and the 2APL language elements and show how this is done. Additionally, the lifecycle of the agent deliberation mechanism is modified to include the normative reasoning performed. Such a modification must take into account cases where not all goals can be reached and possible modifications in the goals or plans already being executed. We also show how this is done.

The rest of the chapter is structured as follows. A formalisation of the planning and normative model is provided in Section 5.1. Section 5.2 describes how the model is implemented by translating it into PDDL and presents the experimental results for this new implementation based on the pizza delivery use case scenario. Section 5.3 gives a detailed description of how the normative reasoner is integrated in the 2APL multi-agent environment and Section 5.4 discusses the overall results and contributions of the chapter.

5.1 Formalisation

As explained in Chapters 1 and 3, our motivating force is how to deal with decision making and planning within normative environments. How can we represent domain knowledge and the normative influence so that the agent simulates a "natural" planning process? Similarly to Section 4.3 we describe our new norm semantics that will lead to the resolution of the issues occurring from the previous approach. Like we did in Section 4.3, we assume again the existence of a normative model NM, defined as in Definition 4.2.

5.1.1 Norms

The operational semantics follow our work on norm lifecycle semantics presented in Section 4.3. To ease operationalisation, our norm representation is a tuple containing a deontic statement and the conditions for norm activating, discharge and violation. Again, the deontic operator of the deontic statement expresses the deontic "flavour" of the norm, the activating, maintenance and discharge conditions are specified as

(partial) state descriptors, denoting the conditions that express when the norm gets activated, violated and discharged. They add operational information to the norm, to simplify the verification and enforcement of the norm.

In essence, when a norm has been activated, has not yet been discharged and the maintenance condition is not true, a violation of the norm happens. Whenever a norm is violated, the norm continues to be active but in addition a norm violation triggers the activation of a *repair norm* (see Figure 5.1). In the figure, after its instantiation, an instance of norm n_1 follows its lifecycle (top part of the diagram). In case a violation occurs (its maintenance condition $f^M_{n_1}$ becomes false) the instance passes to a violation state (V) and at the same time an instance of a second norm n_2 gets activated (middle part of the diagram). The latter now follows its lifecycle independently of the status of the instance of n_1. If a violation of the second instance occurs, then this, in its turn passes to a violation state and at the same time an instance of a third norm n_3 occurs (bottom part of the diagram). A layered dependency of norms, as in Figure 5.1, where one might be considered to repair another (since it gets activated whenever a violation of the previous one occurs) might in theory include infinite norms.

An important restriction explained in Section 4.3.3 is that LTL representation does not allow loops to exist within a transition model. However, in this chapter we lift this constraint as our intention is to be able to create a more flexible lifecycle model which more realistically fits our needs and overcomes the issues explained in the beginning of the chapter. Due to this, it is possible to make a relaxation on the way a violation occurs and persists throughout time. In Section 4.3 a violation occurs and remains throughout the rest of the norm instance's lifecycle, meaning that whenever a violation occurs, it cannot be lifted again. However, this does not have to be the case. In a more relaxed approach a violation might occur whenever maintenance condition ceases to hold and remain until the maintenance condition starts to hold again. This might give rise to a violation of the same instance multiple times within its lifecycle. This is shown in Figure 5.1, where arrows from (V)iolation to (A)ctive were added.

As explained earlier, a repair norm comes to life by the violation, and therefore the rise of the *viol* property, in a norm's lifecycle. Nevertheless, once a norm is violated, we want its repair norm to get activated only when the *viol* first occurs. The problem though is that *viol* is a property that is durative, in the sense that it keeps holding while the norm is violated while, on the other hand, in the activating condition of the repair norm we need to express the exact moment of the violation, in order to prevent the repair norm from being constantly activated whenever a norm is violated. For this reason, we use an additional predicate *prev_viol* to indicate that a norm was previously (in the previous time step) violated. Then, for a repair norm n_{rep} of a norm n, the activating condition could be $viol(n) \wedge \neg prev_viol(n)$. If additionally there exists a repair norm $n_{rep_{rep}}$ of the repair norm, its activating condition would respectively be $viol(n_{rep}) \wedge \neg prev_viol(n_{rep})$. Our main first order language $\mathcal{L}$ from which the activating, discharge and maintenance conditions are constructed is now enriched to include the predicates *viol* and *prev_viol* and we call it $\mathcal{L}'$.

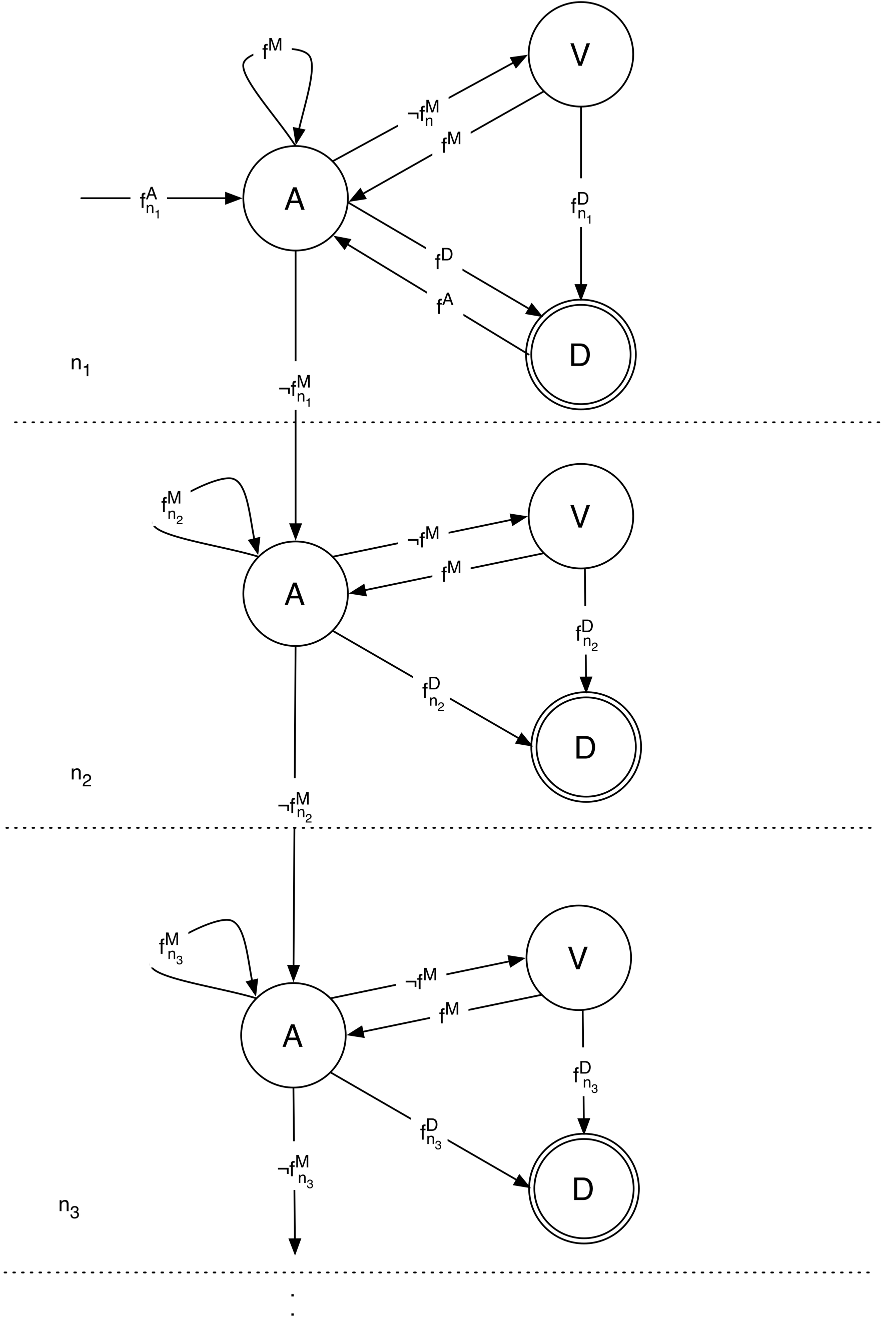


FIGURE 5.1: Layered norms (repairing each other)

We emphasise that, as discussed in [Álvarez-Napagao et al., 2010], this kind of tuple representation including norm activation, discharge and maintenance are as expressive as conditional deontic statements with deadlines (such as the one presented in [Dignum et al., 2005]).

It is also important to note that what changes from the formalisation in Section 4.3 is that the notion of repair is now separated from the norm. Another important feature

of this separation is that once a norm is discharged, it can pass to be instantiated and thus activated anew. This is also an effect of allowing loops within the norm lifecycle, in contrast to the approach of Section 4.3. The added arrow from the (D)ischarge state to the (A)ctive state in Figure 5.1 indicates this change in the formalisation. Below we define the set of norms N:

Definition 5.1. Given a first order language $\mathcal{L}$ and its enriched language $\mathcal{L}'$ with predicates *viol* and *prev_viol*, we define a norm n to be $n = \langle r, f_n^A, f_n^M, f_n^D \rangle$[1], where r is the role it applies to and f_n^A is in $\mathcal{L}'$ and f_n^M, f_n^D are in $\mathcal{L}$.

- The *activating condition* f_n^A specifies when a norm becomes active, i.e. the state of affairs in which the norm is triggered (and must henceforth be checked for completion/violation).
- The *discharge condition* f_n^D specifies when the norm has been discharged, i.e. no longer has normative force.
- The *maintenance condition* f_n^M is needed for checking violations of the norm; it expresses the state of affairs that should hold all the time between the activation and the discharge of the norm.

Following the above representation, the norms of the pizza delivery example would now become three main norms and three repair norms as in Table 5.1. **norm1** would be that one is obliged to follow the direction of the traffic (cannot go the wrong way), **norm2** would be that the delivery of each pizza should be made within the time limit and **norm3** would be that one must drive with less than the maximum speed permitted for each street. The repair norm for each one, **norm1-rep**, **norm2-rep** and **norm3-rep** respectively, would represent the repair condition to be achieved in case of violation.

The way Definition 5.1 is given permits us to construct a hierarchical set of norms, where a norm might be a *primary norm* (meaning that its activeness does not depend on any other norm) or a *repair norm* (meaning that it will get activated whenever a violation of another norm occurs). Such a layered set of norms might include as many norms as desired by the designer. As a rational consequence, the final layer of norms should have the maintenance condition set as `true`, since sequential violations (violation of a norm, and of its repair norm and of the repair norm of the repair norm etc.) should lead to some final repair norm that must always be complied with. Otherwise, the contrary case would result in a system where the final repair norm in a chain of repair norms would have the possibility to be violated, but where no other repair norm would exist to handle it, and therefore, the consequences of its violation would be nonexistent. Note that in Table 5.1 the repair norms **norm1-rep**, **norm2-rep** and **norm3-rep** of the pizza delivery example, the maintenance condition is `true` meaning that they cannot be violated.

[1] In this definition, ***timeout$_n$*** is not present anymore. See Section 5.1.4 for an extended discussion on this.

norm1:Prohibited to go in opposite direction to traffic	
roles	citizen
f_n^A	driving(p, v)
f_n^M	¬movedBetweenJunctions(v, main, street1, street2) or connection(main, street1, street2)
f_n^D	¬driving(p, v)
norm1-rep	
roles	citizen
f_n^A	viol(norm1) ∧¬prev_viol(norm1)
f_n^M	true
f_n^D	fine_paid1(p)
norm2: Deliver on time	
roles	citizen
f_n^A	hasPizzaFor(p, street1, street2)
f_n^M	p_time < goalpizzatime(street1, street2)
f_n^D	pizza_delivered(p, street1, street2)
norm2-rep	
roles	citizen
f_n^A	viol(norm2)∧¬prev_viol(norm2)
f_n^M	true
f_n^D	points_augmented(p)
norm3: Must drive with less than the maximum speed permitted for each street	
roles	citizen
f_n^A	driving(p, v)
f_n^M	¬(vehicle_at(v, street1, street2) and (speedLimit(street1) < vehicleSpeed(v)))
f_n^D	¬driving(p, v)
norm3-rep	
roles	citizen
f_n^A	viol(norm3)∧¬prev_viol(norm3)
f_n^M	true
f_n^D	fine_paid3(p)

TABLE 5.1: Example norms

Below we analyse the deontic interpretation of the newly defined norm. As before, the deontic interpretation of the tuple in Definition 5.1 is done by means of the deontic symbol O.

Definition 5.2. Given a normative model $NM = \langle Roles, Agents, s_0, Actions, Norms, Context \rangle$, the deontic interpretation of a norm $n = \langle r, f_n^A, f_n^M, f_n^D \rangle$ belonging to $Norms$, with agent α in $Agents$, role r in $Roles$, such that $enacts(\alpha, r)$ is:

$$\vdash_{NM} O(E_\alpha[f_n^M] \preceq f_n^D \mid f_n^A)$$

The syntax of the operator O proposed is similar to the obligation operator from Section 4.3.1. This can be informally read as: "*if at some point f_n^A holds, agent α is obliged to see to it that f_n^M is maintained*". Note that in this informal reading we are not dealing with norm instances yet. How we address this issue, along with the semantics of this obligation operator, will be explained in Section 5.1.4.

5.1.2 Norm Instances

We define anew the norm instances, in order to give meaning to the normative planning problem. We use the term θ to denote a substitution. n_α^θ is the norm n with the substitution θ applied to it.

Definition 5.3. Given a normative model $NM = \langle Roles, Agents, IS, Actions, Norms, Context \rangle$ and a norm $n = \langle r, f_n^A, f_n^M, f_n^D, f_n^R, timeout_n \rangle$ in NM, with α in $Agents$, r in $Roles$, such that $enacts(\alpha, r)$ and a substitution θ, we define a norm instance n_α^θ as $n_\alpha^\theta = \langle \alpha, r, \theta f_n^A, \theta f_n^M, \theta f_n^D \rangle$, where:

- θf_n^A is fully grounded, and
- θf_n^M, θf_n^D may be fully or partially grounded.

5.1.3 Norm Lifecycle

We now introduce the properties that will help us with the implementation. As said in the previous sections, the *prev_viol* indicates that there has been a violation in the previous state. We take the properties of Definition 4.8 and adjust them accordingly. Again, we make use of the *Gödelisation* operator $\ulcorner . \urcorner$ [Gödel, 1931] for naming norm instances in our language, with $\ulcorner n_\alpha^\theta \urcorner$ naming the instance n_α^θ.

Definition 5.4. Norm lifecycle predicates

(i) $\langle \pi, 0, \theta \rangle \models active(\ulcorner n_\alpha^\theta \urcorner)$ **if** $\langle \pi, 0, \theta \rangle \models f_n^A \wedge \nexists \theta' : \theta' f_n^D$
and
$\langle \pi, i, \theta \rangle \models \mathbf{X} active(\ulcorner n_\alpha^\theta \urcorner)$ **if** $\langle \pi, i, \theta \rangle \models (\mathbf{X} f_n^A \vee active(\ulcorner n_\alpha^\theta \urcorner)) \wedge \mathbf{X} \nexists \theta' : \theta' f_n^D$

(ii) $\langle \pi,i,\theta \rangle \models viol(\ulcorner n_{\alpha}^{\theta} \urcorner)$ **if** $\langle \pi,i,\theta \rangle \models active(\ulcorner n_{\alpha}^{\theta} \urcorner) \wedge \nexists \theta' : \theta' f_n^M$

(iii) $\langle \pi,i,\theta \rangle \models \mathbf{X} discharged(\ulcorner n_{\alpha}^{\theta} \urcorner)$ **if** $\langle \pi,i,\theta \rangle \models active(\ulcorner n_{\alpha}^{\theta} \urcorner) \wedge \mathbf{X} \exists \theta' : \theta' f_n^D$
and
$\langle \pi,i,\theta \rangle \models \mathbf{X} discharged(\ulcorner n_{\alpha}^{\theta} \urcorner)$ **if** $\langle \pi,i,\theta \rangle \models viol(\ulcorner n_{\alpha}^{\theta} \urcorner) \wedge \mathbf{X} \exists \theta' : \theta' f_n^D$

(iv) $\langle \pi,0,\theta \rangle \models \neg prev_viol(\ulcorner n_{\alpha}^{\theta} \urcorner)$
and
$\langle \pi,i,\theta \rangle \models prev_viol(\ulcorner n_{\alpha}^{\theta} \urcorner)$ **if** $\langle \pi,i-1,\theta \rangle \models viol(\ulcorner n_{\alpha}^{\theta} \urcorner)$ for $i > 0$

Here are some remarks on the new properties: **(i)** remains the same. **(ii)** is simplified to indicate that a norm instance is violated whenever it is active and the maintenance condition is false. By removing the second part of Definition 4.8(ii) we relax the notion of violation as explained in Section 5.1.1, allowing a violation to take place more than once in the lifetime of a norm. That means, that a norm instance violation might occur, and the instance will keep being considered violated as long as it is active and the maintenance condition does not hold. Whenever maintenance condition holds again, the norm is not considered as violated anymore[2]. **(iii)** is also modified to express that the instance is discharged whenever it was active and its discharge condition becomes true, or, whenever it was at a *viol* state and its discharge condition becomes true. **(iv)** from Definition 4.8(iv) though ceases to exist, since in this case a norm cannot reach a state of failure. Instead, we replace it with the semantic notion of *prev_viol*. As we will see in Section 5.2 some of the above rules will serve as basis for the implementation rules for norms within the planning domain.

5.1.4 From Norm to Norm Instances

As in the previous chapter, having formally defined instances we now have the apparatus needed to connect the fulfilment of a norm and the fulfilment of its instances, and give semantic meaning to the operator O proposed in Definition 5.2. This is done in the following definition.

Definition 5.5. Fulfilment of a norm based on the fulfilment of its instances.

[2]We could instead adjust the semantics of violations by modifying Definition 4.8(ii) accordingly:

$\langle \pi,i,\theta \rangle \models viol(\ulcorner n_{\alpha}^{\theta} \urcorner)$ **if** $\langle \pi,i,\theta \rangle \models active(\ulcorner n_{\alpha}^{\theta} \urcorner) \wedge \nexists \theta' : \theta' f_n^M$
and
$\langle \pi,i,\theta \rangle \models \mathbf{X} viol(\ulcorner n_{\alpha}^{\theta} \urcorner)$ **if** $\langle \pi,i,\theta \rangle \models viol(\ulcorner n_{\alpha}^{\theta} \urcorner) \wedge \mathbf{X} \nexists \theta' : \theta' f_n^D$

In this way, the continuity of the violation would be kept from the moment a violation occurred and until the deactivation condition occurs, allowing only one violation to happen throughout the instance's activeness. This would need the appropriate adjustment of the semantics implementation that will follow in Sections 5.2.3 and 5.2.4.

A norm is fulfilled with respect to a path $\pi =< s_0, s_1, \dots, s_m >$ if:

$$\langle \pi, i\rangle \models O(E_\alpha[f_n^M] \preceq f_n^D \mid f_n^A) \text{ iff } \forall\theta : \langle \pi, m, \theta\rangle \models \neg active(\ulcorner n_\alpha^\theta \urcorner)$$

Informally: *the norm is fulfilled with respect to an execution path if, and only if, for each possible instantiation of f_n^A through time, the obligations of the norm instances are not active at the end of the path.*

At this point, we need to explain how the approach of this chapter can be equally expressive to the approach followed in Chapter 4, as far as the time limit for the repair of a norm in case of a violation is concerned. While in the new formalisation we do not explicitly use $timeout_n$ to indicate such a limit, this can be implicitly introduced inside the repair norm (and, in a more generic case, in any norm's condition) in order to indicate time constraints.

One way of doing this (keeping up the notion of the operator $\mathbf{F}_{\leq t}$ of Section 4.2) is by creating special predicates with such semantics that they provide a relative notion of time. That is, predicates that indicate the passing of time starting from a specific time point of the execution. Such a predicate can be called *within* and be implemented so that whenever it is inserted into a norm's condition, it is true during $timeout_n$ timesteps and then becomes false. By applying this, one can use time restrictions within the maintenance condition of a norm. In the case the designer wishes to set a time limit for the norm's compliance, it is then sufficient to add the restriction $within(timeout_n)$ in its maintenance condition, where $timeout_n$ represents a specific time before which the norm (or repair norm) should be accomplished. When using *within* in maintenance conditions, instead of reaching rigid failures (such as the ones occurring from Definition 4.6 and Axiom 4.7), we achieve a more flexible time failure handling, where time limits might produce violations handled by other repair norms.

5.2 Planning with Norms

In this section we propose a methodology to implement the semantics described earlier in this chapter and show how it is done by using the pizza delivery example.

In contrast to Chapter 4, where we opted for the representation of the temporal formulas inside a planner, the basic idea here is to directly implement the rules of Definition 5.4. The main reason for this is that it is not possible to construct a temporal representation for the lifecycle of the norms, since a norm instance might get activated and discharged more than once during the execution, and thus, the cycle occurring by the transition from D(ischarged) to (A)ctive state, as can be observed in Figure 5.1, would prohibit an LTL representation.

In Definition 5.4 the norm lifecycle predicates' calculation depends on the current state of affairs as much as the previous one. This makes it impossible to be directly

passed to a standard planner, since PDDL and similar planning languages only allow effects of actions to depend on predicates of the previous state. In this section, we suggest a solution that uses a standard planning mechanism based on PDDL and *derived predicates*, a PDDL 2.1 feature that allows the expression of axioms that are based on predicates of the current state. Current planners based on PDDL have evolved through several International Planning Competitions (ICAPS) to produce very fast results and provide better features for the design of a planning domain. We will explain more in the following sections.

5.2.1 Plans, Actions, and Plan Cost

In Section 4.6.1 we defined the main elements of classical planning languages. PDDL uses the same notions, with actions having preconditions and effects and a plan being a sequence of actions that generates a path (trajectory). Later versions of PDDL support ADL, derived predicates and typed elements. The complete specification of PDDL 2.1 can be found at [Fox and Long, 2009] and of PDDL 2.2 at [Edelkamp and Hoffmann, 2004].

While PDDL by default includes basic predicates, another kind of predicates, namely *derived predicates* might be defined. These are predicates that are not affected by any of the actions in the domain. Instead, their value in the current state is derived by the value of some basic predicates or other derived predicates through an *axiom* (rule) under the closed world assumption. Then, an instance of a derived predicate (the arguments of which are instantiated with constants) will become true at the current state if it can be derived by the corresponding axiom. Contrary to the basic predicates that may appear in all forms in the definition of actions and goals, derived predicates can only be used non-negated in preconditions, effects, other derived predicates and goals.

Derived predicates are not essential in the definition of a planning domain since they can be incorporated into the action definitions (as complex formulas in the effects) or by adding extra actions that calculate the values of these predicates. Nevertheless, “flattening” a domain in this way will result in a polynomial increase in the size of the domain definition leading to complex, unreadable domains [Thiébaux et al., 2005]. Therefore, derived predicates generally facilitate the elegant and efficient representation of a planning domain. We will use their expressivity to implement our norm lifecycle in Sections 5.2.3 and 5.2.4.

An example of an axiom defining the value of a derived predicate `close-relatives` might be:

```
1 (:derived (close-relatives ?x ?y)
     (or (siblings ?x ?y) (parent ?x ?y) (parent ?y ?x))
3 )
```

Finally, as done previously, by associating actions with a cost function, a plan will have a total cost. The planner will therefore be able to identify the most 'profitable' paths in which norms that might get violated will have their repair norms activated and then dealt with (and these, in their turn, in case they get violated, will have their repair norms activated and then dealt with, etc.). In this way, whether a norm will get violated (and as a consequence have new, repair norms coming to life) becomes part of the planning problem and it is up to the planner to decide which is the most profitable path to follow, according to the cost of the plan.

A domain corresponding to our pizza delivery example containing the action and basic predicate definitions could look like the one in Figure 5.2. This domain is incomplete. Some of the missing parts will be discussed in the next section and the full domain description can be found in Appendix B.3.

```
1  (define (domain pizza_delivery)
     (:predicates
3       (driving ?p ?v)
        (vehicle_at ?v ?street ?street)
5       ...
     )
7
     (:functions
9       (p_time)
        (fine ?p -person)
11      ...
     )
13
     (:derived ...)
15    ...

17   (:action DeliverPizza
       :parameters (?p -person ?street1 -street ?street2 -street)
19     :precondition
        (and
21        (exists (?v)
            (and (driving ?p ?v)
23               (or (vehicle_at ?v ?street1 ?street2) (vehicle_at ?v ?street2 ?street1))))
          (hasPizzaFor ?p ?street1 ?street2)
25         ...)
       :effect
27      (and (pizza_delivered ?p ?street1 ?street2)
             (increase (p_time) 3)
29            ...)
      )
31
      ...
33 )
```

FIGURE 5.2: PDDL pizza delivery domain example

5.2.2 The Normative Planning Problem

We now define the normative problem that occurs under the formalisation presented in this chapter. In Definition 5.5 we gave the notion of a norm's fulfilment with respect

to the fulfilment of its instances. We will use and apply this definition in order to formalise the problem of means-ends reasoning with the influence of norms anew.

Under the current norm semantics, normative planning is formalised as a planning problem in a domain where there are additional norms which acquire normative force through planning paths. Additionally, when an agent fails to comply with them, the norms that handle this takes normative effect (this is done recursively). Having that in mind, the problem then is to find such a plan that the final state achieves the goal, that there are no pending norms[3] and that for all possible violations of norms that might have occurred, the repair norm has been dealt with. We force in this way that all norms which are activated through the different trajectories are either discharged or repaired to make the agent conscious about the effects of each norm violation. In short, the planning problem is defined as finding a plan where:

- The final state achieves the goal(s)[4].
- There are no pending norms (either primary or derived from violations of other norms).

Formally, given a normative model $NM = \langle Roles, Agents, s_0, Actions, Norms, Context \rangle$ and a goal g (which can be a set of goals) we are looking for a plan $\pi = [\alpha_1, \alpha_2, \ldots, \alpha_m]$ generating the trajectory $\tau =< s_0, s_1, \ldots, s_m >$ where all goals are accomplished and additionally there are no pending norms:

- $\langle \pi, m, \emptyset \rangle \models g$
- For all norms $n = \langle r, f_n^A, f_n^M, f_n^D \rangle$ in $Norms$ and all agents α in $Agents$ and roles r in $Roles$ such that $enacts(\alpha, r)$, we have that:

  $\langle \pi, 0 \rangle \models O(E_\alpha[f_n^M] \preceq f_n^D \mid f_n^A)$

As explained in the introductory part of this chapter, it might be counter-intuitive in real-life scenarios to expect for norm instances to be fulfilled according to the operator O, especially whenever these have been violated, raising (a series of) new repair norms to be complied with. For example, a reasonable expectation for a norm that says "*One has to return the book to the library. In the case he does not, he will need to pay an amount equal to the value of the book.*" can be that once the norm gets violated, it "loses" its deontic value, since a new norm which needs to be complied with has been raised. Therefore, while officially the initial norm remains active, an individual needs to only comply with the repair one in order to be considered conformable.

For this reason, we can alternatively use a more relaxed definition of the problem, where we are only interested in "closing" the repair norms that occur from violations, rather than "closing" all the norms. This might make sense in systems where once a

[3]As in Section 4.6.4, by the term "pending" we refer to non-fulfilled norms, with fulfilment being represented by the operator O.

[4]In Section 5.3.2.1 the case when all goals are not reachable, or, reaching them produces excessive violations will be presented.

norm is violated it looses its normative effect, leaving only the consequent norms to take effect. Then the problem would be:

- The final state achieves the goal(s).
- There are no pending norms derived from violations of other norms.

meaning formally that:

- $\langle \pi, m, \emptyset \rangle \models g$
- For all norms $n = \langle r, f_n^A, f_n^M, f_n^D \rangle$ in $Norms$, all agents α in $Agents$, roles r in $Roles$ such that $enacts(\alpha, r)$, and all substitutions θ we have that:
  $\langle \pi, m, \theta \rangle \models \neg active(\ulcorner n_\alpha^\theta \urcorner) \vee viol(\ulcorner n_\alpha^\theta \urcorner)$

  with n_α^θ denoting an instance of norm n

The above occurs as a consequence of the fact that repair norms get activated by the violation of other norms (that contain the predicate *viol*(n) in their activating condition)[5]. Therefore, whenever at the end of the execution a norm is at a violation state this means that a new instance of a repair norm will have been triggered to handle it. As a consequence, the new instance will also need to conform with the above, meaning that it will also have to end up not active or violated at the end of the execution. If it ends up violated, another instance would have been triggered because of the violation that would also need to comply with the above formula etc.

5.2.3 Implementation Rules for Norm Lifecycle

In order to implement the normative reasoner, we need to be able to represent each norm's lifecycle inside the planning domain, in such a way that at every state each of the properties of Definition 5.4 are calculated, for each norm (and the instances that come to life). As already explained, PDDL planning domains consist of definitions of actions that represent the agent's capabilities and modify the domain predicates by adding and deleting facts.

The idea here therefore is to express the properties of Definition 5.4 as predicates within the planning domain. By looking at the rules of the definition, it can be seen that some predicates are expressed in terms of the properties of the current state as well as the previous state. This means that they cannot be directly translated to PDDL predicates, since PDDL only permits each standard domain predicate to have its value calculated over the values of the predicates in the previous state. Therefore, some modifications are necessary. We will modify the rules by creating two types of rules that can be expressed in the planning domain either as derived predicates, or as

[5] We remind the reader at this stage that the final layer of norms would have the maintenance condition set as `true`, since sequential violations should lead to some final repair norm that must always be complied with.

traditional domain predicates. The former will depend on the current state, while the latter will depend only on the previous state.

We choose rules **(i)**,**(ii)** and **(iv)** of Definition 5.4 as a sufficient set to implement. This is because *active* appears in the definition of the normative planning problem (as described in Section 5.2.2) and *viol* and *prev_viol* are the predicates that might appear in the activating condition of a repair norm.

In addition to rules **(i)**,**(ii)** and **(iv)** of Definition 5.4, we define the predicates *inactive*, *complied_with* and *prev_active* to facilitate our implementation[6]. Then, we will have:

By splitting Definition 5.4**(i)** we get:

$(\boldsymbol{\alpha_1})$ $\langle \pi, i, \theta \rangle \models active(\ulcorner n_\alpha^\theta \urcorner)$ **if** $\langle \pi, i, \theta \rangle \models (f_n^A \vee prev_active(\ulcorner n_\alpha^\theta \urcorner)) \wedge \nexists \theta' : \theta' f_n^D$

and

$(\boldsymbol{\alpha_2})$ $\langle \pi, 0, \theta \rangle \models \neg prev_active(\ulcorner n_\alpha^\theta \urcorner)$
$\langle \pi, i, \theta \rangle \models prev_active(\ulcorner n_\alpha^\theta \urcorner)$ **if** $\langle \pi, i-1, \theta \rangle \models active(\ulcorner n_\alpha^\theta \urcorner)$ for $i > 0$

We define *inactive* as:

$(\boldsymbol{b})$ $\langle \pi, i, \theta \rangle \models inactive(\ulcorner n_\alpha^\theta \urcorner)$ **if** $\langle \pi, i, \theta \rangle \models \neg active(\ulcorner n_\alpha^\theta \urcorner)$

We keep Definition 5.4**(ii)** as is:

$(\boldsymbol{c})$ $\langle \pi, i, \theta \rangle \models viol(\ulcorner n_\alpha^\theta \urcorner)$ **if** $\langle \pi, i, \theta \rangle \models active(\ulcorner n_\alpha^\theta \urcorner) \wedge \nexists \theta' : \theta' f_n^M$

We define *complied_with* as:

$(\boldsymbol{d})$ $\langle \pi, i, \theta \rangle \models complied_with(\ulcorner n_\alpha^\theta \urcorner)$ **if** $\langle \pi, i, \theta \rangle \models \neg viol(\ulcorner n_\alpha^\theta \urcorner)$

From Definition 5.4**(iv)** we get:

$(\boldsymbol{e})$ $\langle \pi, 0, \theta \rangle \models \neg prev_viol(\ulcorner n_\alpha^\theta \urcorner)$
$\langle \pi, i, \theta \rangle \models prev_viol(\ulcorner n_\alpha^\theta \urcorner)$ **if** $\langle \pi, i-1, \theta \rangle \models viol(\ulcorner n_\alpha^\theta \urcorner)$ for $i > 0$

It can be seen that $(\boldsymbol{\alpha_1})$, $(\boldsymbol{b})$, $(\boldsymbol{c})$ and $(\boldsymbol{d})$ will be implemented as derived predicates, whereas $(\boldsymbol{\alpha_2})$ and $(\boldsymbol{e})$ as standard predicates. As already explained, derived predicates cannot appear negated in another derived predicate condition or in an action precondition or the goal. Therefore, with the appropriate substitutions, rules $(\boldsymbol{\alpha_1})$, $(\boldsymbol{b})$, $(\boldsymbol{c})$,

[6]The main reason for defining these predicates is the fact that, as mentioned earlier, derived predicates are not allowed to appear negated in action conditions and other predicates. Since in our rules some do appear negated, we need to define the negated version of them as a separate derived predicate.

(d) will become:

(α_1') $\langle\pi,i,\theta\rangle \models active(\ulcorner n_\alpha^\theta \urcorner)$ **if** $\langle\pi,i,\theta\rangle \models (f_n^A \vee prev_active(\ulcorner n_\alpha^\theta \urcorner)) \wedge \nexists\theta' : \theta' f_n^D$
(b') $\langle\pi,i,\theta\rangle \models inactive(\ulcorner n_\alpha^\theta \urcorner)$ **if** $\langle\pi,i,\theta\rangle \models \neg((f_n^A \vee prev_active(\ulcorner n_\alpha^\theta \urcorner)) \wedge \nexists\theta' : \theta' f_n^D)$
(c') $\langle\pi,i,\theta\rangle \models viol(\ulcorner n_\alpha^\theta \urcorner)$ **if** $\langle\pi,i,\theta\rangle \models active(\ulcorner n_\alpha^\theta \urcorner) \wedge \nexists\theta' : \theta' f_n^M$
(d') $\langle\pi,i,\theta\rangle \models complied_with(\ulcorner n_\alpha^\theta \urcorner)$ **if** $\langle\pi,i,\theta\rangle \models inactive(\ulcorner n_\alpha^\theta \urcorner) \vee \exists\theta' : \theta' f_n^M$

Now **(α_1')**, **(b')**, **(c')** and **(d')** can be directly implemented as derived predicates in PDDL. Since **(α_2)** and **(e)** depend on the previous state, they are implemented as standard domain predicates added to the effects of every action. The first part of **(α_2)** and **(e)** is redundant in PDDL, as by default all predicates are false in the initial state, unless stated otherwise. The second part of each of the two rules is then expressed as follows:

(α_2') $\langle\pi,i-1,\theta\rangle \models active(\ulcorner n_\alpha^\theta \urcorner) \Rightarrow \langle\pi,i,\theta\rangle \models prev_active(\ulcorner n_\alpha^\theta \urcorner)$ for $i > 0$
$\langle\pi,i-1,\theta\rangle \models inactive(\ulcorner n_\alpha^\theta \urcorner) \Rightarrow \langle\pi,i,\theta\rangle \models \neg prev_active(\ulcorner n_\alpha^\theta \urcorner)$ for $i > 0$

(e') $\langle\pi,i-1,\theta\rangle \models viol(\ulcorner n_\alpha^\theta \urcorner) \Rightarrow \langle\pi,i,\theta\rangle \models prev_viol(\ulcorner n_\alpha^\theta \urcorner)$ for $i > 0$
$\langle\pi,i-1,\theta\rangle \models complied_with(\ulcorner n_\alpha^\theta \urcorner) \Rightarrow \langle\pi,i,\theta\rangle \models \neg prev_viol(\ulcorner n_\alpha^\theta \urcorner)$ for $i > 0$

Therefore, we have now ready the full set of rules **($\alpha_{1,2}'$)**-**(e')** to be implemented in PDDL. This is seen in Table 5.2. The way these are implemented within the planning domain is as follows: For each norm (both primary and repair), a separate, norm-specific predicate (made distinct by the addition of the norm name at the end) *active*, *inactive*, *complied_with*, *prev_active*, *viol*, *prev_viol* is defined. For (repair) norms where the maintenance condition is `true`, predicates *viol*, *complied_with* and *prev_viol* are omitted, since their values can be pre-calculated through the rules as `false`, `true` and `false` respectively.

For each norm, rules **(α_1')**, **(b')**, **(c')**, **(d')** are expressed through a derived predicate in the planning domain, expressing the conditions f_n^A, f_n^M and f_n^D in PDDL terms[7]. An example of the representation of the *viol* predicate of rule **(c')** for **norm1** of the pizza delivery example as the derived predicate `viol_norm1` can be the following:

```
1 (:derived (viol_norm1 ?p -person ?v -vehicle)
    (and (active_norm1 ?p ?v)
3        (not (not (exists (?main -street ?street1 -street ?street2 -street)
                    (and (movedBetweenJunctions ?v ?main ?street1 ?street2)
5                        (not (connection ?main ?street1 ?street2)))))) ))
```

[7]Such a translation should be straightforward in PDDL 2.1 where all the necessary elements such as ADL, functions, existential and universal quantifiers are supported.

As derived predicates:
(α_1') $\langle \pi,i,\theta \rangle \models active(\ulcorner n_\alpha^\theta \urcorner)$ **if** $\langle \pi,i,\theta \rangle \models (f_n^A \vee prev_active(\ulcorner n_\alpha^\theta \urcorner)) \wedge \nexists \theta' : \theta' f_n^D$
(b') $\langle \pi,i,\theta \rangle \models inactive(\ulcorner n_\alpha^\theta \urcorner)$ **if** $\langle \pi,i,\theta \rangle \models \neg((f_n^A \vee prev_active(\ulcorner n_\alpha^\theta \urcorner)) \wedge \nexists \theta' : \theta' f_n^D)$
(c') $\langle \pi,i,\theta \rangle \models viol(\ulcorner n_\alpha^\theta \urcorner)$ **if** $\langle \pi,i,\theta \rangle \models active(\ulcorner n_\alpha^\theta \urcorner) \wedge \nexists \theta' : \theta' f_n^M$
(d') $\langle \pi,i,\theta \rangle \models complied_with(\ulcorner n_\alpha^\theta \urcorner)$ **if** $\langle \pi,i,\theta \rangle \models inactive(\ulcorner n_\alpha^\theta \urcorner) \vee \exists \theta' : \theta' f_n^M$

As domain predicates:
(α_2') $\langle \pi,i-1,\theta \rangle \models active(\ulcorner n_\alpha^\theta \urcorner) \Rightarrow \langle \pi,i,\theta \rangle \models prev_active(\ulcorner n_\alpha^\theta \urcorner)$ for $i > 0$
$\langle \pi,i-1,\theta \rangle \models inactive(\ulcorner n_\alpha^\theta \urcorner) \Rightarrow \langle \pi,i,\theta \rangle \models \neg prev_active(\ulcorner n_\alpha^\theta \urcorner)$ for $i > 0$
(e') $\langle \pi,i-1,\theta \rangle \models viol(\ulcorner n_\alpha^\theta \urcorner) \Rightarrow \langle \pi,i,\theta \rangle \models prev_viol(\ulcorner n_\alpha^\theta \urcorner)$ for $i > 0$
$\langle \pi,i-1,\theta \rangle \models complied_with(\ulcorner n_\alpha^\theta \urcorner) \Rightarrow \langle \pi,i,\theta \rangle \models \neg prev_viol(\ulcorner n_\alpha^\theta \urcorner)$ for $i > 0$

TABLE 5.2: PDDL domain implementation rules

Additionally, for each norm, rules **(α_2')**, **(e')** are implemented by a domain predicate, the value of which will be passed from one state to another through the respective rule. Since a domain might include several actions that pass the system from one state to another, each action should contain the norm's rules in its effects (followed by the obvious `and` connector). An example of the representation of the *prev_active* predicate of rule **(α_2')** for **norm1** of the pizza delivery example as the domain predicate `prev_active_norm1` inside any action's effects can be the following:

```
(forall (?p -person ?v -vehicle)
2  (when (active_norm1 ?p ?v) (prev_active_norm1 ?p ?v)))
(forall (?p -person ?v -vehicle)
4  (when (inactive_norm1 ?p ?v) (not (prev_active_norm1 ?p ?v))))
```

At this point we would like to focus on an issue that occurs when applying the rules with a repair norm. As stated, a repair norm $nr = \langle r, f_{nr}^A, f_{nr}^M, f_{nr}^D \rangle$ would contain $viol(n_\alpha^\theta) \wedge \neg prev_viol(n_\alpha^\theta)$ in its activating condition f_{nr}^A, to indicate that an instance might get activated whenever there is a violation of a norm n and the norm n was not being violated before. The problem with this activating condition is that the *viol* predicate that it contains, which is defined as a derived predicate, cannot appear in rule **(b')** as part of the activating condition, since the activating condition f_{nr}^A is part of a negated formula. The solution to this is to substitute the activating condition in rule **(b')** as follows:

$\langle \pi,i,\theta \rangle \models inactive(\ulcorner nr_\alpha^\theta \urcorner)$**if**

(from **(b')**)

$\langle \pi,i,\theta \rangle \models \neg((f_{nr}^A \vee prev_active(\ulcorner nr_\alpha^\theta \urcorner)) \wedge \nexists \theta' : \theta' f_{nr}^D)$**if**

(substituting f_{nr}^A)

$\langle \pi,i,\theta \rangle \models \neg((viol(n_\alpha^\theta) \wedge \neg prev_viol(\ulcorner n_\alpha^\theta \urcorner) \vee prev_active(\ulcorner nr_\alpha^\theta \urcorner)) \wedge \nexists \theta' : \theta' f_{nr}^D)$**if**

$\langle \pi,i,\theta \rangle \models (\neg(viol(\ulcorner n_\alpha^\theta \urcorner) \wedge \neg prev_viol(\ulcorner n_\alpha^\theta \urcorner) \vee prev_active(\ulcorner nr_\alpha^\theta \urcorner)) \vee \exists \theta' : \theta' f_{nr}^D)$**if**

$\langle \pi,i,\theta \rangle \models ((\neg viol(\ulcorner n_\alpha^\theta \urcorner) \vee prev_viol(\ulcorner n_\alpha^\theta \urcorner) \wedge \neg prev_active(\ulcorner nr_\alpha^\theta \urcorner)) \vee \exists \theta' : \theta' f_{nr}^D)$**if**

(from $(\boldsymbol{c'})$)

$\langle\pi,i,\theta\rangle \models (((\neg(active(\ulcorner n_{\alpha}^{\theta}\urcorner) \wedge \nexists\theta' : \theta' f_n^M) \vee prev_viol(\ulcorner n_{\alpha}^{\theta}\urcorner)) \wedge \neg prev_active(\ulcorner nr_{\alpha}^{\theta}\urcorner)) \vee \exists\theta' : \theta' f_{nr}^D)$**if**

$\langle\pi,i,\theta\rangle \models ((((\neg active(\ulcorner n_{\alpha}^{\theta}\urcorner) \vee \exists\theta' : \theta' f_n^M) \vee prev_viol(\ulcorner n_{\alpha}^{\theta}\urcorner)) \wedge \neg prev_active(\ulcorner nr_{\alpha}^{\theta}\urcorner)) \vee \exists\theta' : \theta' f_{nr}^D)$**if**

(from $(\boldsymbol{\alpha_1'})$)

$\langle\pi,i,\theta\rangle \models ((((\neg((f_n^A \vee prev_active(\ulcorner n_{\alpha}^{\theta}\urcorner)) \wedge \nexists\theta' : \theta' f_n^D) \vee \exists\theta' : \theta' f_n^M) \vee prev_viol(\ulcorner n_{\alpha}^{\theta}\urcorner)) \wedge \neg prev_active(\ulcorner nr_{\alpha}^{\theta}\urcorner)) \vee \exists\theta' : \theta' f_{nr}^D)$

"Flattening" a formula so as not to contain any negative predicates of the type *active*, *inactive*, *viol* or *complied_with* is always possible, by applying all substitutions. Such a substitution is necessary to happen before any of the rules $(\boldsymbol{\alpha_{1,2}'})$-$(\boldsymbol{e'})$ is implemented inside the PDDL domain. An example of the representation of the *inactive* predicate of rule $(\boldsymbol{c'})$ for **norm2-rep** of the pizza delivery example as the derived predicate `inactive_norm2-rep` can be the following:

```
  (:derived (inactive_norm2-rep ?p -person ?street1 -street ?street2 -street)
2   (or (and (or (or (not (and
                          (or (hasPizzaFor ?p ?street1 ?street2)
4                             (prev_active_norm2 ?p ?street1 ?street2))
                      (not (pizza_delivered ?p ?street1 ?street2))))
6                 (not (and (hasPizzaFor ?p ?street1 ?street2)
                            (> (p_time) (goalpizzatime ?street1 ?street2)))))
8             (prev_viol_norm2 ?p ?street1 ?street2))
          (not (prev_active_norm2-rep ?p ?street1 ?street2)))
10      (points_augmented ?p)))
```

The full code for the implementation of the scenario can be found in Appendix B.3.

5.2.4 Implementation Rules for Normative Planning Problem

As explained in Section 5.2.2, the objective is to find a plan $\pi = [\alpha_1, \alpha_2, \ldots, \alpha_m]$ generating the trajectory $\tau = < s_0, s_1, \ldots, s_m >$ such that it reaches the goal(s) and leaves no norms active. Therefore, given a norm set N, at the final state m, for all norms $n = \langle r, f_n^A, f_n^M, f_n^D \rangle \in N$, all agents α such that $enacts(\alpha, r)$ and all substitutions θ:

$\langle\pi, m, \theta\rangle \models \neg active(\ulcorner n_{\alpha}^{\theta}\urcorner)$

The above is equivalent to stating that given a norm set N, at the final state m:

For all norms $n = \langle r, f_n^A, f_n^M, f_n^D \rangle \in N$, all agents α such that $enacts(\alpha, r)$ and for all substitutions θ:

$\langle\pi, m, \theta\rangle \models inactive(\ulcorner n_{\alpha}^{\theta}\urcorner)$

Therefore, for every norm, we should add to the PDDL problem file such a statement. Taking for example **norm1**, **norm2**, **norm3** and their repair norms **norm1-rep**, **norm2-rep**, **norm3-rep** of the pizza delivery domain, we would add the following code inside the goal of the problem file:

```
(and
  (forall (?p -person ?v -vehicle) (inactive_norm1 ?p ?v))
  (forall (?p -person ?v -vehicle) (inactive_norm1-rep ?p ?v))
  (forall (?street1 -street ?street2 -street) (inactive_norm2 ?street1 ?street2))
  (forall (?street1 -street ?street2 -street) (inactive_norm2-rep ?street1 ?street2))
  (forall (?p -person ?v -vehicle) (inactive_norm3 ?p ?v))
  (forall (?p -person ?v -vehicle) (inactive_norm3-rep ?p ?v))
)
```

5.2.5 Computational Overhead

Generally, planners' performance depends on the implementation of the planning algorithm. Most current planner implementations apply forward search algorithms and variations of the enforced hill climbing algorithm (e.g. Metric-FF [Hoffmann and Nebel, 2001] and SGPlan [Hsu and Wah, 2008]). However, as we will see in Section 5.2.6.2 where our experimental results are presented, we use the option of applying the algorithm A* that is provided with Metric-FF to ensure optimal results. A*'s complexity in general depends on the heuristic, with the worst case being exponential. Metric-FF applies A* search and as heuristic it uses the cost of the relaxed plan as an estimation for the remaining cost.

Given that we "force" the planner to calculate a great number of instances for the three derived predicates *active*, *inactive*, *viol*, *complied_with* and the domain predicates *prev_active*, *prev_viol* there is definitely an increase of the domain size and therefore of the plan graph. Additionally, the calculations over those states depend highly on the number of instances of each norm, since the checks performed on every step are done for all possible instances.

Having the two in mind, it is not possible to make concrete calculations on the overall overhead in the execution time. Nevertheless, since the normative influence comes mainly from the activeness of the norm instances and the fulfilment of the repair norms after every violation (which normally has as a consequence some action being executed in order to satisfy the repair norm's discharge condition), the plan cost should work as a pretty good heuristic. In other words, norms act as guidelines in the planner's execution, since they "lead" the execution towards less costly solutions, where norms are chosen to be violated or complied with in the "cheapest" possible way.

5.2.6 Results

In this section we briefly introduce the tools we have employed in order to create normative planners (the Metric-FF planner), detail our experimental results and compare them to the ones of the previous chapter.

5.2.6.1 Tools

Metric-FF[8] is a domain independent planning system extending the Fast Forward planner [Hoffmann and Nebel, 2001]. Fast-Forward (abbreviated FF) is a forward chaining heuristic state space planner. In fact, the basic principle of FF is that of HSP, first introduced by Bonet et al. in [Bonet et al., 1997]. The FF creators use a novel local search method, called enforced hill-climbing. Enforced hill-climbing is a hill-climbing procedure that, in each intermediate state, uses breadth first search to find a strictly better, possibly indirect, successor. If local search fails, then it skips everything done so far and switches to a complete best-first algorithm that simply expands all search nodes by increasing order of goal distance evaluation.

Metric-FF handles a combination of PDDL 2.1 and ADL, and is implemented in C. An important feature of the latest version, Metric-FF 2.1, is that it handles derived predicates. Its algorithm can handle effects and constraints on linear functions over numerical state variables and favours minimisation of a given cost function. Like most other planners, it works by *relaxing* of the original planning problem, that is, by ignoring all the delete effects in the effects of the actions. The occurring task is called *relaxed task*, while a plan for a relaxed task is called *relaxed plan*. If configured to favour speed and efficiency over quality, Metric-FF uses relaxed plan length as the goal distance estimation - the number of steps in the relaxed plan - in enforced hill-climbing, combined with *helpful actions pruning* in the search, a method that only considers applicable actions that add at least one goal at the lowest layer of the relaxed solution (and switch to a complete weighted A* search in case it fails to find a solution). Otherwise, if configured to favour minimisation of a given cost function, Metric-FF performs a standard weighted A* search where the cost of the relaxed plan is seen as an estimation for the remaining cost. The weight parameters of the search can be given in the command line (weight of the plan quality, that is, its length and cost versus the total solving time). The cost function may contain linear expressions over numerical state variables, under the condition that it can be translated to action costs. The cost of the relaxed plan, in that case, will be the total of all costs of the actions belonging to that plan.

[8] http://fai.cs.uni-saarland.de/hoffmann/metric-ff.html

5.2.6.2 Execution Results

In this section we present our experimental results, based on the pizza delivery example adjusted to the new norm semantics. We use the same experimental scenarios (with id 1-7) and the same computer and operating system as in Section 4.6.6. The complete code for the example can be found in Appendix B.3.

Not surprisingly, the outcome in every case is the same plan with the same cost. This makes absolute sense since both approaches under the same conditions and weight for the three factors **`p_time`**, **`penalty_points`** and **`fine`** should conclude with the same decision. This approach though, clearly outperforms the one described in Chapter 4, as all executions are accomplished within a second or less.

When executing Metric-FF we used the option to apply a weighted A* search, since hill climbing algorithms are not optimal. The execution results can be seen in Table 5.3. Apart from the execution time, the number of states visited by Metric-FF can be seen.

id	evaluated states	execution time (in secs)
1	133	1.28
2	635	4.96
3	664	4.83
4	246	1.96
5	40	0.99
6	136	1.32
7	124	1.29

TABLE 5.3: Execution Results

Below we provide some interesting observations about the results:

- We are confident that we could have implemented the approach of this chapter in TLPLAN. We would have done so by substituting derived predicates with predicates of type `def-defined-predicate`. Nevertheless, we did not try it as we felt that 1) a generic and widely used planning language such as PDDL and a planner processing it would be more appropriate for our needs and 2) TLPLAN is slightly outdated (with its latest version being of 2008) compared to Metric-FF (with its latest version, Metric-FF 2.1, being of 2012).
- As explained in Section 4.6.6, the main drawback of the approach of Chapter 4 is the inefficiency, when it comes to complex control rules, such as our LTL formulas for the lifecycle of the norms. Furthermore, the execution time seems to worsen disproportionally to the number of instances occurring throughout the calculation of a plan. However this can hardly be attributed to TLPLAN, since it has been shown to perform equally well to its contemporary planners [Bacchus and Kabanza, 2000]. We strongly believe that the overhead occurs mainly as a consequence of the control rule checking at every search state. This has been a

strong motivation towards getting rid of the LTL norm lifecycle representation and moving in the direction of the layered repair norms.

- The approach of this chapter in combination with the use of Metric-FF has given much better results. We attribute this to the fast algorithm of the planner. The use of derived predicates generally adds an additional cost to the execution time, but this is still manageable in our experiments. However, it has to be noted that the use of `exists` and `forall` formulas (deriving from the $\nexists$ and $\exists$ in rules **(a)-(d)** of Table 5.2) in the PDDL derived predicates can significantly slow down the planning time, especially whenever there are many variables in the maintenance and discharge conditions of a norm that have not been already substituted (variables that they have not in common with the activating condition).
- Metric-FF provides the option to favour execution time over ''preciousness" of the plan as well as to time limit the search. This can be especially useful in order for the agent to have the option to always get a plan and not wait too long to have the best result. In real time environments, where speed is an important factor, this is an essential quality.

5.3 Connecting the Normative Reasoner with the 2APL BDI Core

With an implementation of a normative planner in hand, the next step is to integrate it within an existing agent framework. In this section we introduce the norm-aware planning component described in the previous sections, which can create plans influenced by the norms, in the agent's deliberation cycle (more concretely in its practical reasoning step) . We have chosen **2APL** [Dastani, 2008], a modular BDI-based agent programming language, as the basis for our norm-aware agents.

There are a few compelling reasons for this choice against other frameworks described in Section 2.1.1.2. Jadex [Braubach et al., 2005] does not have a logic-based syntax that would enable transformation from/to statements in a classical planning language. Jason [Bordini and Hübner, 2006] has the disadvantage that it operates in old versions of Java (1.5) and does not provide an easy way to modify the deliberation cycle available. Furthermore, to our best knowledge, 2OPL [Dastani et al., 2009a] does not have a stable implementation yet.

On the other hand, 2APL provides, in a clear and simple logic-based syntactical representation, the main elements of the BDI architecture: beliefs, goals and plans. Furthermore, the framework supports a representation of actions (represented as the agent's belief updates as we will see in section 5.3.1) with preconditions and effects, facilitating the mapping of the planning domain actions from/to 2APL. This makes it easy to make transformations from 2APL to planning domains. The language in addition supports abstract actions, which are useful for interpreting and handling the

normative planner's output. 2APL also includes the concept of events for notifying environmental changes which are useful to indicate points in time when the normative reasoner should be re-executed. Equally important is the fact that 2APL has an adjustable deliberation process. That means that it can easily enable the modification of the agent's cycle and add the functionality of our normative planner. Finally, the software has a recent implementation available and is under constant improvement by the developers.

5.3.1 2APL

2APL [Dastani, 2008] is an agent-oriented platform, based on the BDI model, that allows the implementation of both reactive and proactive agents within multi-agent systems. Being of a modular nature, it provides the option to plug-in cognitive components in its modules and also the tools to define the agents of a multi-agent system and the environment(s) in which the agents act. In 2APL, the programmer can specify the basic elements of an agent, that is, the beliefs, goals, plans, actions and events received by external entities. The actions can be of several different types, such as belief updates, external actions, or communication actions. Additionally, the platform provides extra rules for each agent (providing specifications for plans), which enable it to decide which actions to perform and how to act in case of failure. A screenshot of the main 2APL user interface can be seen in Figure 5.3.

5.3.1.1 2APL Elements

2APL comprises of the following elements:

- **Beliefs**: Beliefs are Prolog facts and form a belief base. Any Prolog construction can be used in the belief base. An example of a belief is '`at(home).`'.
- **BeliefUpdates**: Belief updates update the belief base of an agent when executed. BeliefUpdates contains the specification of the belief update actions (see Belief Update Actions).
- **Goals**: Goals are Prolog facts and form a goal base. An example of a goal could be '`at(work).`'.
- **Plans**: A plan is a program to be executed. In general plans consist of actions. Amongst others, some types of actions are:
  - **Belief Update Actions**: Actions that update the beliefs of the agent when defined through the BeliefUpdates. These actions are specified in terms of preconditions and postconditions (effects). An agent can execute a belief update action if the precondition of the action is derivable from its belief base, and its execution modifies the belief base so that that the effect of

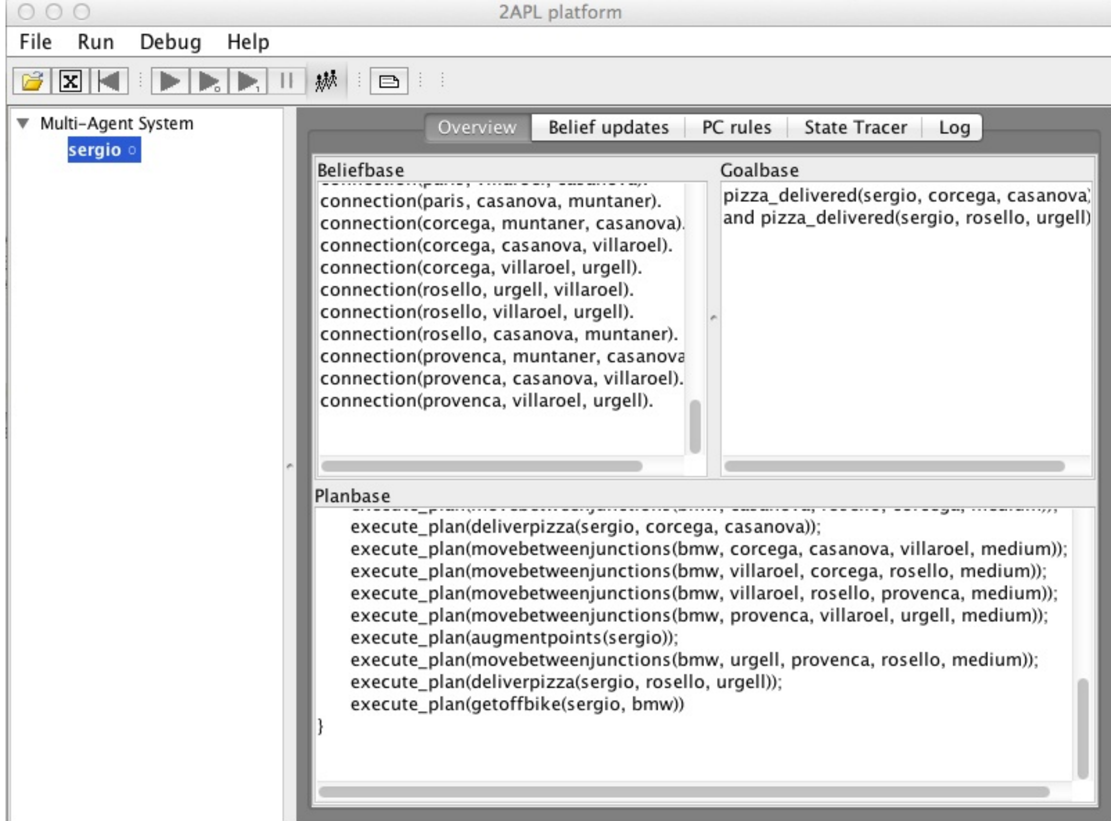


FIGURE 5.3: Screenshot of the 2APL environment

the action is derivable from the belief base after[9]. An example of such an action would be '{at(P,X)} MoveFromTo(P,X,Y) {at(P,Y), not at(P,X)}'.

- **Communication Actions**: Actions passing messages between agents.
- **External Actions**: Actions that affect the environment and are denoted with the symbol '@' to indicate that they modify the environment. An example of an external action could be '@myworld(go(helen,home,work))', where myworld is the name of the environment.
- **Abstract Actions**: Actions that are defined through a Procedural (PG) Rule. An abstract action encapsulates a plan. The execution of an abstract action causes this action to be replaced with the plan from the body of the rule in which it is defined. Therefore, when trying to execute an abstract action in a plan, this will instantiate a 'sub-plan' and substitute itself with that. An example of an abstract action might be 'my_abst_act(P,X,Y)'.

[9]While Belief Update Actions modify the belief base of the agent, this does not necessarily mean an environment modification. The agent's belief base might partially reflect the agent's perception of the environment, however, it is kept separately for each agent and solely represents its internal knowledge. An environment modification on the other hand might only occur, as we will see, through the execution of an External Action. Therefore, whenever an agent performs a Belief Update Action which reflects some environment change, this should be accompanied by a relevant External Action.

- **PG-rules (Planning Goal Rules)**: These rules generate a plan to be executed when some conditions are met. They consist of three entries, a head, a condition and a body. Each PG rule contains in its body a specification of a plan, which will be adopted, if specific goals and beliefs (found in the head and condition of the rule respectively) are derivable. An example of such a rule could be 'at(P,work)<-alarm | {WakeUp(P) ; MoveFromTo(P,_,work) ; @myworld(go(P,_,world))}'.
- **PC-rules (Procedural Rules)**: These rules are activated and create plans when 1) the agent receives a message sent by another agent, 2) an event, generated by the external environment, occurs, or 3) there is an abstract action to be executed. They too consist of three entries, a head, a condition and a body. The head might contain a message, an event, or an abstract action, the condition the beliefs under which the plan will be adopted and the body of the plan to be adopted. In the case of an abstract action, the PC rule therefore identifies the plan (its body) that will take the place of the abstract action within a larger plan. An example of such a rule defining the abstract action my_abst_act can be 'my_abst_act(P,X,Y)<-true| {MoveFromTo(P,X,Y);@go(P,X,Y)}'.
- **PR-rules (Plan Repair Rules)**: These rules are provided for the case that the execution of an agent's plan fails.

5.3.1.2 2APL Deliberation Cycle

The mental state of the 2APL agent is formed by the beliefs, goals, plans and reasoning rules. The deliberation cycle specifies what the agent's behaviour will be, for example the execution of an action or the application of a reasoning rule, by executing a deliberation process. It can be therefore seen as the interpreter of the agent program, as it indicates specific steps to be applied in some particular order. The 2APL deliberation cycle can be seen in Figure 5.4. The following steps are executed in it:

- Rules are checked and any new plans produced are added to the plan list to be executed (PG-rules).
- The first action of each plan is executed. After these actions, goals are queried in the belief base. Reached (derivable) goals in the base are considered achieved and are removed from the goal base and any plan triggered by these is removed from the plan base.
- Failed plans are handled (PR-rules).
- Messages are processed (PC-rules).
- External events are processed (PC-rules).

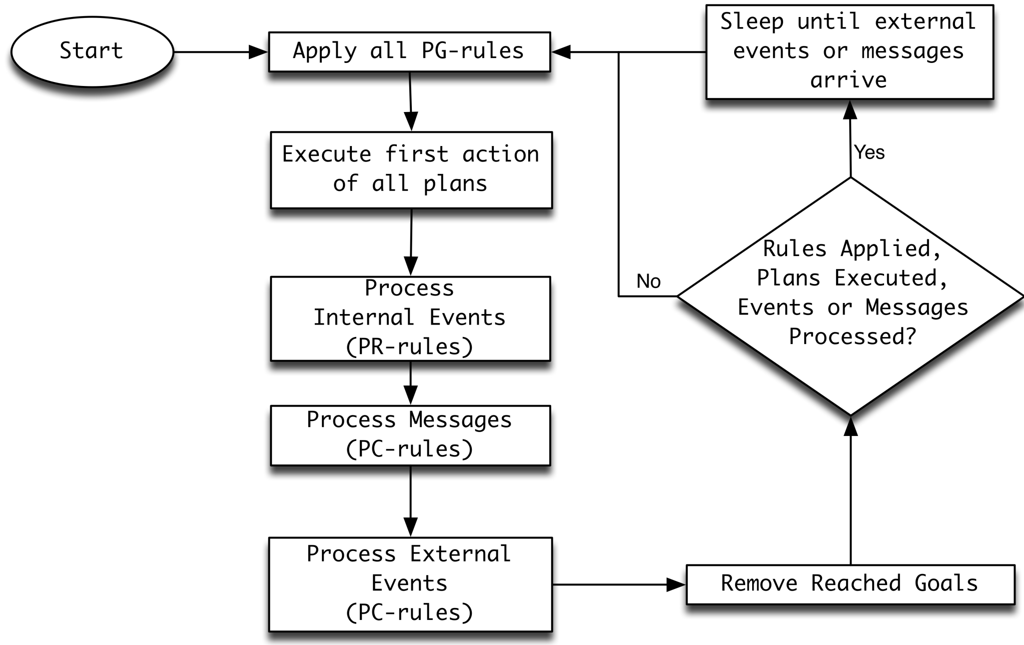


FIGURE 5.4: The 2APL deliberation cycle

5.3.2 2APL with Embedded Norm-Aware Planner

One particular problem which is not properly tackled is norm integration with the reasoning phase in the BDI cycle (that is, to decide how to reach a goal while taking into account the norms). In our approach norms are not seen as restrictions, but as guidelines in the practical reasoning. Unlike most frameworks, our approach takes into consideration the norms during the planning phase, this is what we call "norm-oriented planning". Therefore, we introduce in the agent's deliberation cycle a norm-aware planning component that can create plans influenced by the norms. We implement the practical reasoning step in the BDI cycle via the normative planning problem and the Metric-FF planner presented in Section 5.2.

5.3.2.1 Modified 2APL Lifecycle

The modification of the 2APL lifecycle (seen in Figure 5.5) is made in such a way that the PG-rules, which are primarily responsible for generating plans, are excluded. This is because we are interested in an automatic, on-demand generation of plans, taking into consideration the norms and not having rules forcing the means-ends reasoning in 2APL's original form. The new process is as follows.

1. If there are goals to be achieved in the goal base, 2APL prepares the inputs for the planner making certain transformations to produce an appropriate planning domain. Section 5.3.3.1 shows how this is done.

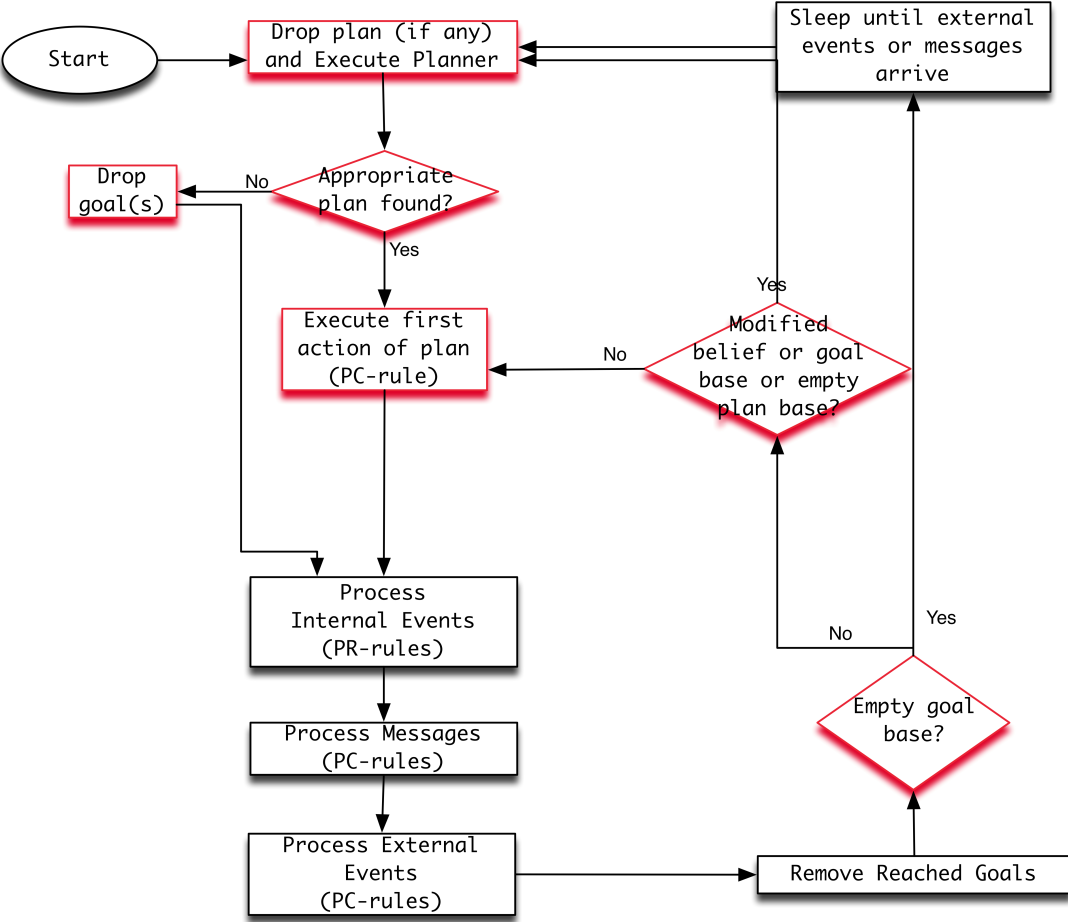


FIGURE 5.5: The modified 2APL deliberation cycle

2. The normative planner generates a plan, given the beliefs, belief updates and goals of the agent. If a plan is found and this plan is considered to be acceptable in terms of numbers of violations or other criteria that the agent might have, the agent puts it in its plan base to be executed. In an alternative case where the plan is seen unsuitable, one or more goals might be dropped, in order to lead the reasoning mechanism to find more effective plans for less goals than the original set.

3. An action of the plan, transformed appropriately from the planner's output to 2APL terms is executed (PC-rule). Section 5.3.3.2 shows how the planner actions are transformed to and seen as PC-rules.

4. Failed plans are handled (PR-rules). As explained, these rules are triggered whenever a plan fails. In our case, with the plan being a sequence of abstract actions, a repair rule for each of these can be defined, making sure that some other process (for example a substitution for the specific action) takes place, or even, that one or more goals are dropped.

5. Messages and external events are processed as in the original deliberation cycle (PC-rules).
6. Goals that have been reached are removed from the goal base.
7. If there are no more goals left, the agent sleeps until some new event occurs or some message arrives.
8. If there are still goals to be reached, then, whether there are changes in the belief and goal base or whether the plan base is left with no plan is checked. In the case that changes in the belief or goal base have indeed occurred, the agent drops any plan that was being executed and goes back to step 1 of the cycle, to execute the planner.
9. In the case where no changes have occurred in the belief and goal base or the plan has finished its execution, and therefore no plans are left in the plan base, then the cycle goes to step 3 of the cycle to continue executing the plan.

The evaluation of the plan found in step 2 is optional and up to each agent to do. A criterion could be some threshold on the number of the violations that might happen during the execution of the plan, or a high plan cost, which would be prohibiting and non-profitable for the agent. In the case where an agent decides that the plan is not suitable, according to that criteria, then some goals might be (temporarily) dropped in order for the reasoner to produce plans for a subset of the original goals. This requires an advanced reasoning process, in order to decide which subset of goals might be dropped, since dropping one goal randomly might result in the rest of the agent's goals not being meaningful anymore. We consider such a process domain-dependent and therefore out of the scope of the thesis.

5.3.2.2 General Architecture

Figure 5.6 depicts the general architecture of the system. Norms can be obligations or prohibitions and are accompanied by repair norms (similarly defined norms) in case they are breached. Norms are expressed according to Definition 5.3 and fed to the planner as explained in Section 5.2, influencing the planning mechanism within the system. In combination with utility functions over the actions, the system computes the most profitable trajectory concluding with a state of the world where no norms awaiting settlement exist.

The next sections explain the details of the integration of the normative planner with the 2APL framework.

5.3.3 Adapting Inputs Between 2APL and the Normative Planner

In this section we explain how the normative planner interacts with the 2APL platform and how the inputs and outputs of the planner are transformed from/to the 2APL

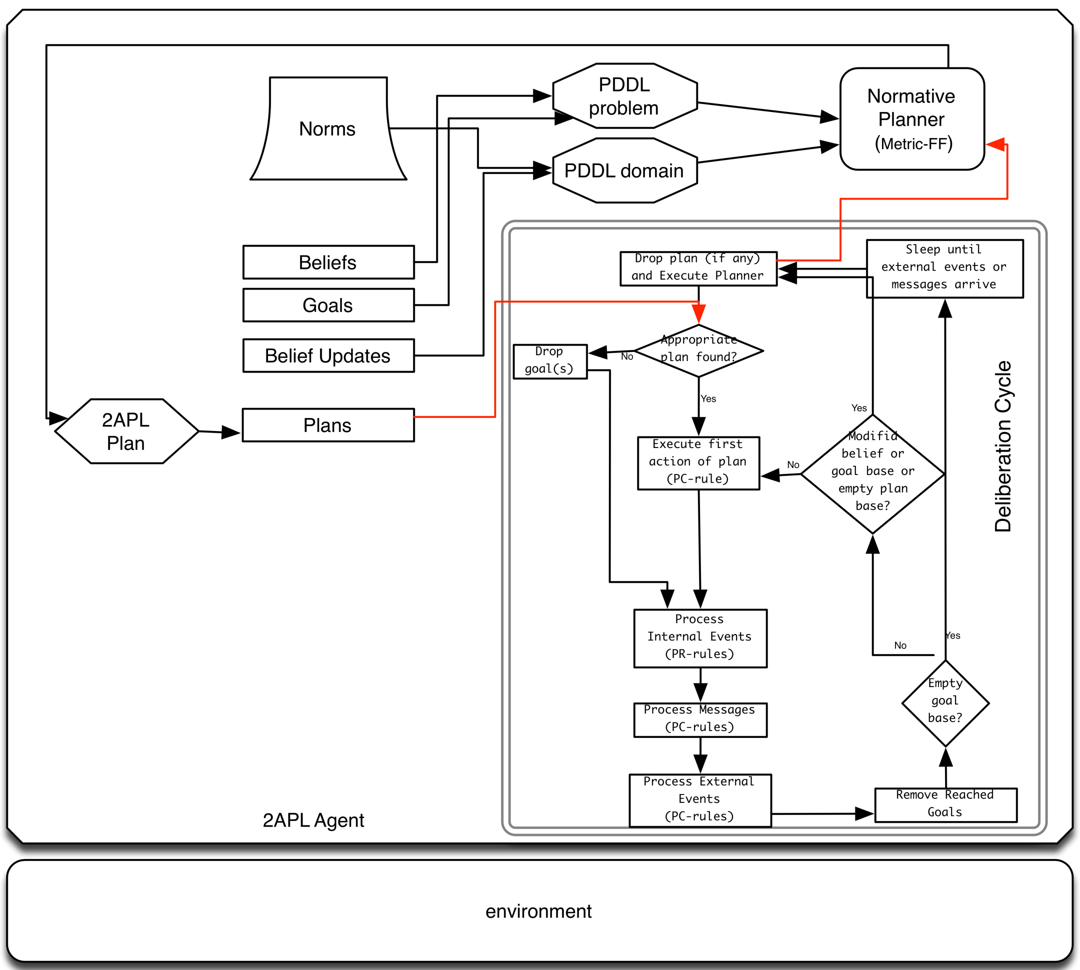


FIGURE 5.6: 2APL Planner Architecture

agent language.

5.3.3.1 Adapting Inputs from 2APL into Normative Planner

The belief updates (see section 5.3.1) of the 2APL agent (where the first part is the precondition, the second is the name and the third is the effect) can be considered the capabilities (actions) of the agent. Each of these will be transformed into a PDDL domain action. In our pizza delivery example, the belief updates of the agent will be `MoveBetweenJunctions`, `DeliverPizza`, `AugmentPoints`, `PayFine1` and `PayFine3`.

The beliefs of the agent are transformed to the PDDL problem instance. The beliefs include the initial values of the factors `p_time`, `penalty_points` and `fine`. The translation from 2APL to PDDL is pretty straightforward as most atoms in the Prolog facts in 2APL are structures simple enough to produce the predicates for the initial state in the PDDL planning problem. A special 2APL belief **fun** in the belief base

<table>
<tr><td>2APL</td><td><pre>BeliefUpdates:

{
 driving(P,V) and
 (vehicle_at(V,Street1,Street2) or
 (vehicle_at(V,Street2,Street1))) and
 hasPizzaFor(P,Street1,Street2) and
 p_time(T)
}
DeliverPizza(P,Street1,Street2)
{
 pizza_delivered(P,Street1,Street2),
 pizza_delivered(P,Street2,Street1),
 not hasPizzaFor(P,Street1,Street2),
 not hasPizzaFor(P,Street2,Street1),
 p_time(T+3), not p_time(T)
}</pre></td></tr>
<tr><td>PDDL Domain</td><td><pre>(:action DeliverPizza
 :parameters (?p ?street1 ?street2)
 :precondition
  (and
   (exists (?v) (and
    (driving ?p ?v)
    (or (vehicle_at ?v ?street1 ?street2)
        (vehicle_at ?v ?street2 ?street1))))
  (hasPizzaFor ?p ?street1 ?street2)
  (checked-norm)
  (checked-norm-rep))
 :effect
  (and
    (pizza_delivered ?p ?street1 ?street2)
    (pizza_delivered ?p ?street2 ?street1)
    (not (hasPizzaFor ?p ?street1 ?street2))
    (not (hasPizzaFor ?p ?street2 ?street1))
    (increase (p_time) 3)
    (not (checked-norm))
    (not (checked-norm-rep)))
)</pre></td></tr>
</table>

TABLE 5.4: Belief Updates Transformation

(as described in Section 4.6.4) represents the function that the agent considers as the best optimisation setting for its decisive factors. This is translated into the planning problem's metric to be minimised and expressed in the PDDL `metric` statement.

Finally, the goals of the agent are directly transformed to the goals in the PDDL problem instance. The normative objectives (explained in Section 5.2.2) are automatically added to the goals, that is, that there is no active obligation coming from violated norms at the end of the execution of the plan.

An example of belief updates, beliefs (including **fun**) and goals in 2APL and how they

are translated to the PDDL domain and problem can be seen in Tables 5.4, 5.5 and 5.6[10].

<table>
<tr><td>2APL</td><td>

```
Beliefs:

 driving(sergio, bmw).
 vehicle_at(bmw, muntaner, provenca).
 vehicleSpeed(bmw, 10).
 hasPizzaFor(sergio, corcega, casanova).
 goalpizzatime(corcega, casanova, 28).
 hasPizzaFor(sergio, rosello, urgell).
 goalpizzatime(rosello, urgell, 19).
 p_time(0).
 fine(sergio,0).
 penalty_points(sergio,0).
 movedBetweenJunctions(bmw, muntaner, rosello, provenca).
 fun(1*((p_time/30)+1)+
     1*((penalty_points(sergio)/10)+1)+
     1*((fine(sergio)/60)+1)).
```

</td></tr>
<tr><td>PDDL Problem</td><td>

```
(:init
 (driving sergio bmw)
 (vehicle_at bmw muntaner provenca)
 (= (vehicleSpeed bmw) 10)
 (hasPizzaFor sergio corcega casanova)
 (= (GOALPIZZATIME corcega casanova) 28)
 (hasPizzaFor sergio rosello urgell)
 (= (GOALPIZZATIME rosello urgell) 19)
 (= (p_time) 0)
 (= (fine sergio) 0)
 (= (penalty_points sergio) 0)
 (movedBetweenJunctions bmw muntaner rosello provenca)
)

(:metric
  minimize (+ (+ (* 1 (+ 1 (/ (p_time) 30)))
                 (* 1 (+ 1 (/ (penalty_points sergio) 10))))
                 (* 1 (+ 1 (/ (fine sergio) 60)))))
```

</td></tr>
</table>

TABLE 5.5: Beliefs Transformation

5.3.3.2 Adapting the Normative Planner Output into 2APL

As mentioned in Section 5.3.3, the belief updates of the agent are considered to be the actions that the agent is capable of performing at different times. Consequently, the normative planner will return a plan, consisting of some of those actions (in probably a slightly modified format, as for example Metric-FF returns small letter action names in contrast to the original 2APL belief updates, which follow the 2APL language case sensitivity).

[10]The whole code can be found in Section B.4 of Appendix B, in Figure B.11 and Figure B.12

2APL	Goals: pizza_delivered(sergio, corcega, casanova) and pizza_delivered(sergio, rosello, urgell)
PDDL Problem	(and (pizza_delivered sergio corcega casanova) (pizza_delivered sergio rosello urgell))

TABLE 5.6: Goals Transformation

We implemented the 2APL agent in such a way that whenever a plan is produced by the normative planner, the previous plan is dropped and all the actions in the newly produced plan are inserted into the 2APL agent's plan base, each wrapped within an abstract action "execute_plan()". We do this to ensure that during the actual execution of the action, multiple things can happen. In principle, those will be the execution of the belief update action and possibly external actions that update the environment accordingly. For this reason, for each belief update, an abstract action encapsulating the belief update and the possible external actions corresponding to it, needs to exist in the PC-rules.

An example of the specification of an abstract action for the DeliverPizza(P, Street1, Street2) belief update can be seen in Figure 5.7. Line 5 indicates the execution of the belief update and line 6 indicates the environment action that corresponds to the same belief update.

```
PC-rules:

execute_plan(deliverpizza(P, Street1, Street2))<-true|
{
 DeliverPizza(P,Street1,Street2);
 @pizzaworld(deliverPizza(P,Street1,Street2))
}
```

FIGURE 5.7: Abstract action represented by a PC-rule

As a consequence, whenever the outcome of the normative planner includes deliver pizza(sergio, corcega, casanova), this will be inserted into the plan base as execute_plan(deliverpizza(sergio, corcega, casanova)).

5.3.4 Running the Normative Agent in 2APL

In this section we give a brief description of how 2APL with the normative planner integrated is executed. We developed and executed 2APL through the Eclipse Integrated

Development Environment for Java[11]. As mentioned, 2APL supports the design of the environment which the agents share and with which the agents will interact. We have developed a simple environment with a GUI depicting a map (Figure 5.8) that shows at which location the agent is found, the route it is following while moving on the grid and the delivery of every pizza. At the same time, 2APL outputs the actions performed by the agent, including the ones that pay fines or augment its penalty points.

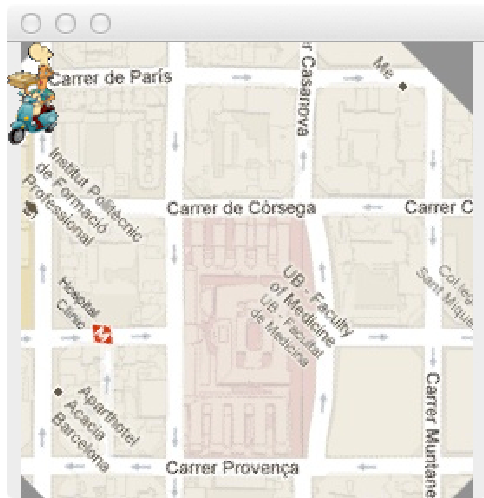


FIGURE 5.8: 2APL environment GUI. Delivery Map

As explained in Section 5.3.3.2 the agent's belief updates are seen as its capabilities (possible actions to be performed) and are represented by an abstract action in a procedural rule (PC-rule). This abstract action performs the belief update as well an environment action corresponding to it. Therefore, each action needs to have a direct implementation in the environment, in order for its actual effects to take place in the environment.

We have implemented the pizza world environment to the include actions "`@enter`", "`@moveBetweenJunctions`", "`@deliverPizza`", "`@getOffBike`", "`@payFine1`", "`@augmentpoints`", "`@payFine3`" which correspond to the agent's belief updates. "`@enter`", "`@moveBetweenJunctions`" and "`@deliverPizza`" interact with the graphical map by placing, moving the agent and showing the delivery of a pizza respectively. The rest have a dummy implementation within the environment as in our example they do not affect the environment's state of affairs (and therefore could be prevented from being executed within the abstract action; still we include them as it would allow possible extensions of the example).

An extra component called "**PlanStep**" responsible for transforming all the elements necessary to the planning domain, executing Metric-FF and transforming the resulting plan (if any) back to 2APL abstract actions was created. This component was introduced into the agent cycle with the modification of the 2APL cycle as shown in Section 5.3.2. We have designed our conceptual metamodel of Section 3.2 using

[11]https://www.eclipse.org

the Eclipse Modelling Framework (EMF)[12] technology which facilitates model-driven development of Java applications. The norms are defined independently by the agent designer as an instance of our framework metamodel in Eclipse. They are processed by **PlanStep** and together with the 2APL belief updates, beliefs and goals create the planning domain and problem to be passed to the planner. Finally, Metric-FF is integrated into the agent platform by adding the Metric-FF executable to the 2APL framework[13].

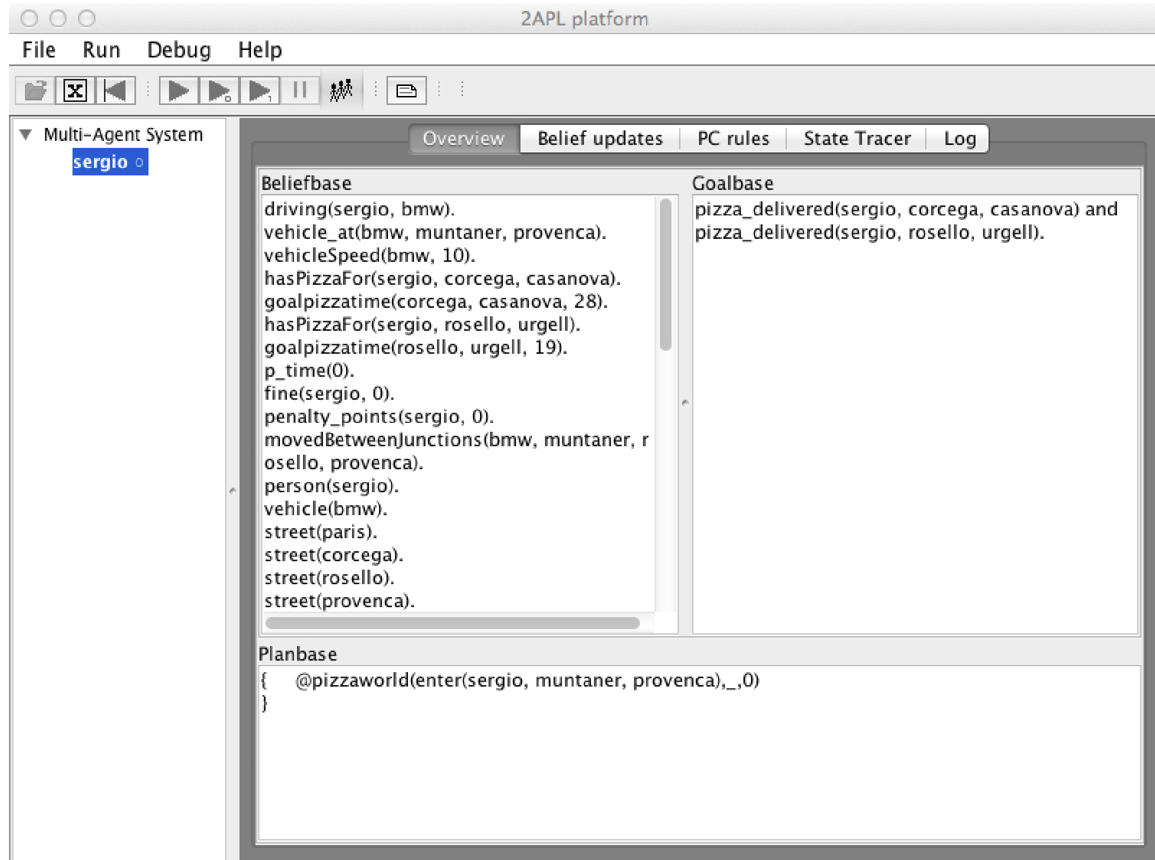


FIGURE 5.9: 2APL initial configuration for the pizza delivery example

We executed the experiments of the pizza delivery example with the 8 different configurations in the integrated version of 2APL together with the planner. The time was counted from the moment **Start Deliberation** button ▶ is hit until the agent has reached the point where no more goals exist in its goal base (that is, both pizzas have been delivered). The results can be seen in Table 5.7. As expected, the execution time rises (roughly by a fixed amount of time of 2.0-2.5 seconds) mainly because of two factors. Firstly, the graphical environment burdens the execution with the obvious overhead and secondly, 2APL is implemented in Java, meaning that the deliberation phase is executed through a Java process, which adds to the total time. However, these factors are expected and cannot be avoided whenever dealing with complete

[12] http://www.eclipse.org/modeling/emf/

[13] In our case we simply made it part of the Eclipse project but alternatively, it could be part of the 2APL executable jar.

and functional frameworks that might offer the possibility for graphical displays and additional features.

id	planner execution time (in secs)	2APL execution time (includes planning, in secs)
1	1.28	3.02
2	4.96	6.73
3	4.83	6.84
4	1.96	4.85
5	0.99	3.4
6	1.32	3.03
7	1.29	3.05

TABLE 5.7: Execution Results in the 2APL agent environment

5.4 Discussion

5.4.1 Contributions and Extensions

The strength of our normative framework lies in the realistic and fully functional representation of the normative reasoning problem. It uses semantics which have been implemented by several (PDDL) planners while it only adds a relatively small overhead to the planning process caused by the introduction of the derived predicates calculating the norms' status. Existing planners allow for several features to be integrated into the planning process (durative actions, etc.) and the framework could be expanded to include time propositions within the definition of the norm. It can be further extended towards utility functions over the states of the world instead of the actions. This alternative allows the representation of preferences of a state over another, which can be even closer to the human reasoning process.

Emphasis is given to the agent's welfare attributes and how these are influenced by the execution of a plan with respect to a set of norms. Given the fact that two agents might vary in the importance that they give to individual factors, different outcomes concerning the achievement of a goal are expected. Referring to the pizza delivery scenario for example, in the case of an upright agent who really avoids getting fined, he might prefer to stay obedient to the regulations at the expense of his time, while in the case of a less committed agent, he might risk his financial status in order to deliver faster.

Further to this, multiple norms may be in conflict and an agent must make informed choices. For example, if an agent has an obligation to bring about a goal g, but bringing about goal g means that it will have to violate one of two other existing norms, then

the agent faces a conflict that will need a decision to be made, over which norm to be violated (or even not to follow the initial obligation in the first place). In such a case, the agent, being aware of the consequences that the violation of each one could have given the circumstances, might be able to make an informed decision based on situational criteria rather than always producing the same outcome.

In this chapter, an integration of our normative reasoner within a multi-agent environment was presented. A strength of the design of our reasoner is that it can be integrated into and interact with other environments in a similar fashion. The requirement for this to happen is for the environment to support agents that possess and can process individual capabilities and that function in a goal-driven manner. Assuming that the agent's capabilities can be translated into planning domain actions, normative reasoning can take place and return a plan for the agent to handle. The integration into an agent's cycle is not always trivial, but in most cases it will be a one-step process occurring during the cycle of the agent whenever its beliefs get modified or whenever an agent has no plans left to execute or whenever the plan being executed fails.

A main drawback of our approach is that whereas our framework works well in environments where the consequences of an agent's conformance to normative restrictions are known or can be estimated, it is not always the case that this is feasible. In many cases the effect of a violation cannot be known in advance, or might even be nonexistent (not always does driving too fast imply a fine). Our approach assumes that there is an instant global enforcing mechanism and cannot for the moment handle probabilistic effects of norm deviation. A future extension of this framework could include such probabilistic outcomes, where for example a norm's violation does not always lead to the instantiation of a repair norm, but instead, a repair/penalty norm gets raised according to some probabilistic function, as would happen in a real-world situation, where in order for a sanction to be assigned and applied, a monitor needs to have observed it.

We have made the mappings between BDI and planning systems in a similar way as Sardina has in [Sardina et al., 2006], which focuses on the integration of the HTN planning process into the BDI architecture, without reasoning about norm-based restraints. An essential difference however, when evaluating the planner's contribution to the agent's means-ends reasoning in Sardina's research, is that the plan library is seen as the method library (consisting of prescriptions of how HTN tasks are decomposed into subtasks) in the HTN planner. We, on the other hand, completely discard the agent pre-fabricated plans or plan templates (the corresponding 2APL PG-rules) and deploy a planning mechanism that computes plans based purely on the agent's capabilities seen as primitive domain actions.

There exist several works that consider norms with some *repair*, *punishment* or *sanction* taking over or becoming effective whenever a violation occurs [López y López and Luck, 2002; Dastani et al., 2009a; Villatoro et al., 2011; Alechina et al., 2012].

Other frameworks on the other hand keep track of the number of violations occurring throughout an execution path [Oren et al., 2011]. Both approaches are used to directly or indirectly determine the validity or preciousness of plans. The problem with the use such "heuristics" is that they do not always reflect real situations where the consequences of a norm violation are neither direct, neither weigh the same, nor do they directly correspond to a quantitative measure of a simple, for example, measurable sanction. We manage to successfully address this problem by: 1) Defining repair norms that come to life as a result of a violation, and 2) Building a layer of such repairs, since one might not be always certain that an agent will desire and decide to comply with the repair steps designed to be taken in case of a violation and might need a chance to breach this process too.

As a final remark, we have found the idea of Hindriks and van Riemsdijk about using timed transition systems [Hindriks and van Riemsdijk, 2013] interesting, since such systems would allow the incorporation of more elaborate time formulas and deadlines. However, in the planning community the closest to this would be PDDL 2.1 [Fox and Long, 2009] which allows temporal planning by assigning duration to actions. We believe that our framework could be extended to include complex forms of time and norms containing formulas referring to complex time constraints.

5.4.2 Revisiting Requirements

Table 5.8 summarises the requirements covered by the framework presented in this chapter.

Req.	Description	Status	Justification
R1.1	Deliberative, means-end, norm-oriented reasoning mechanism	✓	We apply a planning mechanism that receives norms in the form of path control rules.
R1.2	Decision making process guided by user preferences	✓	The planning mechanism supports action costs and tries to maximise the overall value of the plan, according to the criteria set by the agent.
R1.3	Goal driven decision making	✓	The planner performs its planning algorithm trying to achieve some goals. These goals are the agent's goals.
R1.4	Agent capabilities specification accommodated by framework	✓	The agent capabilities can be seen as the possible actions to be performed. The action language used supports action descriptions.
R1.5	Adjust in case of relevant environment change	✓	The modification of the 2APL deliberation cycle takes into consideration the case where there is a change in the environment (perceived as modification of the belief base of the agent). Such a case leads to the calculation of a new plan, with respect to the new information.
R1.6	Norm conflict toleration	✓	The formalisation used takes into account norm conflicts. The normative planning problem definition comes up with a solution possibly containing norm violations, but that is most profitable for the agent according to the criteria set.

R2.1	Full domain/environment definition	✓	The domain representation occurs from the beliefs and belief updates of the 2APL agent being translated into PDDL, a language that permits a full representation of domain information and actions. Additionally, the external environment is represented in the 2APL as a separate entity. In this, the user can define what the impact of the agent's behaviour to the environment is.
R3.1	A well defined normative model allowing the clear and unambiguous interpretation of norms on a operational level	✓	The formalism presented in this chapter provides a solid normative model and the definitions of norm compliance and of the normative problem provide a functional interpretation of norms.
R3.2	Mechanisms for agent behaviour monitoring of norms	✓✗	Again, in this chapter, a mechanism for norm monitoring has not been implemented but the formalism is created in a way that monitoring is possible to be implemented.
R4.1	Agent-oriented architecture	✓	Our formalism addresses and is integrated within an agent-oriented architecture, where agent capabilities and preferences are seen as and translated to domain actions. The resulting plan serves as a plan to be executed by the agent.
R4.2	Open standards support	✓	We provide a generic mechanism to support normative reasoning. We represent our normative model on the Eclipse Modelling Framework (EMF) which is openly accessible. Furthermore, the actions representation in Metric-FF is in PDDL, a widely used standard in the planning community.
R4.3	System platform-independent model	✓	We provide a generic mechanism to support normative reasoning. We represent our normative model on the Eclipse Modelling Framework (EMF) which enables various transformations and thus interoperability with other frameworks.
R4.4	Strong focus on semantics at domain, agent context, ontology, normative level	✓	The formalisation (Normative Model, norm fulfilment, norm instances) provides a clear and functional understanding of all these elements.
R4.5	Tool Support for norm and domain representation	✓	A tool for norms representation is provided, since the metamodel is defined in Eclipse Modelling Framework (EMF). The agent's capabilities are expressed in the 2APL framework as belief updates and are translated into the domain actions. The agent's preference criteria are also written in the 2APL language as part of the agent description.
R4.6	Support for multiple standards and extensibility	✓	We provide a generic mechanism to support normative reasoning. We represent our normative model on the Eclipse Modelling Framework (EMF) which enables various transformations and thus interoperability with other frameworks.
R4.7	Soft real time performance	✓	The experimental results of this chapter's approach have shown a significant improvement (a few seconds in the worst case) to the previous chapter's results, while the integration with the 2APL framework adds a stable non-significant overhead. Therefore, we can assume that the soft real time performance criterion is fulfilled.
R4.8	Reasoner's response time priority over optimality	✓	Metric-FF provides the option to assign a specific weight for the configuration option passed to the planner. This weight acts as a value factor over the plan length and cost against the execution cost in time. Such an option can be used to compute solutions that are faster but not optimal.
R4.9	Validity	✓	In our case, validity translates to the the planner providing a correct solution to the planning problem specified [Ghallab et al., 2004]. Given that the Metric-FF algorithm ensures the correctness of the plan returned, this requirement is fulfilled.

Table 5.8: Requirements Analysis

Chapter 6

Conclusions

In Section 1.1 the main question to tackle in this thesis was presented:

"How to model an autonomous, goal-driven agent that is able to take the environment's normative influence into account in its decision making?"

In order to provide answers and practical solutions to it we have pursued three general objectives that are essential when dealing with practical materialisation of norm-aware systems: 1) straightforward connection between the deontic level and the operational semantics, 2) the formalisation of explicit norm instances, and 3) the establishment of clear semantic interpretation across implementation domains.

We have done so through various steps. We first created a conceptual normative framework which includes a generic, platform and language-independent representation of norms and its formalisation (both structural and dynamics). Special focus has been given to modelling both the lifecycle for norms and their (multiple) instantiations. We have also defined an architecture for norm-aware agents aligned with our framework, which is supported by abstract metamodels that ease its implementation for different platforms and data formats.

We then built a connection between deontic statements and temporal logics, and between temporal logics and temporal constraints that can be translated to control rules within a (planning) domain. Using these rules we implemented a normative planner. This has made it possible to generate plans that comply with the restrictions imposed by the norms. Further to this, we have explored how, by using costs (possibly derived from multiple factors) for different state transitions, the compliance (or the consequences in the case of non-compliance) of a plan to a set of norms relates to the measurement of its quality. We have achieved in this way a mechanism that constructs plans that not only comply with the restrictions imposed by the norms but also do so in an optimal way.

Due to computational limitations of this first approach, we extended the norm semantics to include multiple layers of reparation norms and restructured the decision making problem. Using a life-like scenario, we have provided execution results indicating

the usefulness and efficiency of our approach. We finally integrated the normative planner into the 2APL multi-agent framework and showed how this process is done.

6.1 Contributions

Mechanical or "formal" reasoning has been explored by philosophers and theorists since the ancient times. By the 90's Artificial Intelligence had developed advanced and complex mechanisms to deal with rapidly evolving and even non-deterministic environments, based on notions the roots of which, can be found in legal and social sciences as well as economics.

This thesis sets as its objective one of the most interesting areas of research in Artificial Intelligence, that is reasoning and decision making while taking norms into account. It applies knowledge representation, reasoning and planning, all of which form central domains of AI. More concretely, representation of objects, relations, actions, events, time, causes, effects and states is attempted and put into practice.

Our work stems from the fact that while regulation of agent's behaviour has become a necessity in current multi-agent environments, little work on practical reasoning mechanisms within normative environments providing both a formalisation and an implementation, exists. In this thesis we have aimed at filling this gap. Our contributions have been several within the computational area of AI:

- By combining areas such as norm-aware agent systems, planning and decision making, we have brought to light a new perspective on normative reasoning within agent systems. In this, agents are no longer limited by a list of predefined plans, but instead, use a planner to calculate paths towards their goals, while at the same time allowing the occurring norm instances to influence the calculation of these paths.
- We have achieved the design of normative agents that dynamically interpret, apply and reason over norms (whether to deviate from predetermined behaviour), becoming in this way norm-autonomous.
- We put into practice a realistic approach which captures the essence of decision making under the influence of norms. While human deliberation process is not fully known, researchers believe that the human deliberation process, based on a goal expectation, is a product of a set of alternative options and a set of selection criteria, qualitatively evaluated [Wang and Ruhe, 2007]. Given that humans often deploy a complex planning mechanism to explore such alternative options, we consider our approach to be rather intuitive and close to the human decision making process.
- We have shown the usefulness of our framework within a scenario that applies in real life.

A strength of our normative reasoner lies in the fact that it can be applied to many different BDI (and other types of) systems. The main necessity for this is that agents' capabilities can be represented by a planning language such as PDDL. Having designed the reasoner's elements through an MDE approach, one can map an agent's elements to our model and integrate the reasoning mechanism into a bigger, multi-agent system in a fairly easy manner.

It is also important to note that, despite the fact that this thesis mainly focuses on the agent perspective of normative reasoning, the norm semantics of our thesis in Chapters 4 and 5 (Sections 4.3 and 5.1, respectively) have been defined in a way that can be useful for both the individual norm-agent operating in the environment and for the norm enforcement mechanisms. This is vital to ensure that in the same environment both the agents and the enforcement mechanism share the same semantics for norm activation, discharge and violation.

We consider the reduction of our norm formalisation to known deontic logics in Section 4.4 to be a significant contribution to the state of the art. Our in-depth analysis and mapping to formalisms that are already well established within the scientific world proves that our normative system fits well the popularly accepted by the research community concepts of *what norms are* and *how they might be interpreted*. We are unaware of other similar computational representations and analyses towards existing deontic logics.

Another interesting contribution of our work is that both norm representations in Chapters 4 and 5 are inspired by and reflect the human deliberation mechanism, as far as behavioural restrictions are concerned. We believe that having a task to execute or a goal to reach, humans tend to consider normative influences with respect to personal and social interests by taking into account several non-conscious welfare attributes or preferences. In our case, these are represented in absolute numbers in the planning problem, but some alternative could be that a preference function between possible situations is defined instead.

Our norm formalism takes into account a very significant issue when dealing with normative environments: norm conflict handling. This is done through the way that our normative definition problem is given, in Sections 4.6.3 and 5.2.2 respectively, for each formalisation. The definition allows, according to the agent's own criteria, to decide for two or more conflicting norms, which one(s) would be more profitable to have violated (and let the repair handle). Unlike other frameworks which deal with it through an offline verification mechanism, we incorporate this into the problem and let the planning mechanism resolve it.

Finally, while we did not create a complicated mechanism that deals with failures, we showed how the reasoner can be incorporated into a typical BDI reasoning cycle. Our modification of 2APL deliberation cycle happens on two levels: 1) agent belief updates are seen as potential capabilities (actions) under the appropriate circumstances and their actual execution can take place following a simple or complex procedure

containing the effects over the agent as well as the environment and 2) there is no longer the need of plan production rules; instead, this is substituted by the normative planning mechanism. The suggested cycle can cope with possible occurring events and focuses on reaching the agent's goals. In the case where not all goals can be achieved, then, some are dropped and the process continues. We believe that our integration methodology is generic enough. In the same way our normative reasoner is introduced into the 2APL deliberation process, it can also be integrated into other BDI implemented systems, provided that easy modification of the cycle is permitted.

6.2 Revisiting Claims

In Section 1.5 we provided a set of claims for this thesis. In this section, we analyse them and validate them one by one.

C.1. *Throughout the agent's deliberation process, means-ends reasoning can be performed by invoking a planner instead of using pre-compiled plans*. We use this assumption based on the observation that the pre-compiled plans can instead be substituted by plans that are produced at runtime. In Chapter 5 we demonstrated how a planner is used and integrated within an existing agent system, 2APL, that performs a deliberation process.

C.2. *The planning process can be influenced with norms*. This is the central topic of our thesis. We achieved this objective by defining and implementing the normative reasoning problem through the reduction of norm semantics to a planning domain and using a planner to receive norm-aware plans, in both Chapters 4 and 5.

C.3. *Standard Deontic Logic is not sufficient to capture practical, complex aspects of norms' functionality*. In Section 2 we provided a background on agent-based and normative systems, while in Section 2.2.2.2.3 we explained the specification of the most commonly used language by theorists, Standard Deontic Logic. We explained the difficulties and the ambiguities that might occur when trying to formally capture norm semantics with the use of this logic on a practical level.

C.4. *Temporal logic might be used to capture the norms' lifecycle*. In Chapter 4 we gave a formal definition and made a mapping of the norm's lifecycle to LTL temporal formulas, which in their turn, reflect suitable norm-aware execution paths.

C.5. *Extend existing agent framework with norms*. In Chapter 5 the 2APL agent platform was extended to support our normative framework and planning mechanism and an implementation explaining how the 2APL connection from/to the normative planner is done.

C.6. *Actions with costs can be used for modelling domain and user preferences*. In both approaches of Chapters 4 and 5 the agent's capabilities were mapped to actions

with preconditions and effects over the environment. These, along with functions that assign specific weights to each action performed based on various factors, form the domain.

C.7. *A norm might be defined on an abstract level, allowing a flexible representation like a template for the construction of concrete instances*. In both formalisations of Chapters 4 and 5 the norm definition is abstract and made in such a way that its activating condition can give rise to various instances throughout time via the possible substitutions of its variables.

6.3 Extensions

While we cannot claim that our approach covers all aspects of normative multi-agent environments, we consider it a first step that could have further extensions towards more advanced implementation. The work presented could be extended in various directions. This section is dedicated to some directions in which our work could be taken to in order to be better incorporated to the agent's scientific field.

6.3.1 General Extensions

In Section 3.2 we presented an architecture which includes several interactions with an ontology. Such an ontology not only describes the elements used within the framework but also provides domain and range axioms, class expressions, qualified cardinality restrictions, nested class expressions and more features. As a future step such an ontology could be interconnected to the metamodels and the framework in order to express more complex relationships between the framework elements and enabling reasoning, this time on a semantic level.

In an environment where several agents coexist the need for practical reasoning where parallel activities are involved provides a clear time efficiency. Therefore, an interesting feature to be explored is the treatment of concurrency, where actions are allowed to be executed in parallel. This will then lead to plans that contain concurrent interacting processes.

In modern technology, where actions are mostly represented by services, often it happens that there is no single service capable of performing a task, but there are combinations of existing services that could do so. For this reason a representation of action synthesis (creation of a composite action) is needed. Therefore, an extension of our framework could take this into account and reason over a domain which consists of simple and complex actions.

In the framework presented we make the assumption that the goal(s) of an agent are pre-specified. Still this is not always realistic in a constantly changing environment.

This work could be taken one step further, exploring the goal features at a meta-reasoning process where the *dedication*, *urgency* and *confidence* of an agent over a goal can change over time [Vecht, 2009].

In our work, we have left out constitutive norms. Nevertheless, it could be extended to include constitutive norms and/or institutional power, as defined by Jones et al. in [Jones and Sergot, 1996; Jones et al., 2013]. This would possibly require an additional level of reasoning before the planner being executed in order to establish relationships and empowerment of different entities as well another level for the "counts-as" statements to be translated to rules supported by our planning mechanism, such as derived rules in the planning domain.

Another interesting line of research is to extend our work towards multiple agent decision making. While one agent considers its personal settings in order to create a plan that satisfies its needs, the issue becomes more complex when dealing with more agents and social metrics. Such an extension might become subject to many questions:

- Are all agents able to perform the same actions?
- Do norms equally apply to all agents, or does every agent need to conform to a separate set of norms? In the first case, are norms instantiated separately for every agent? Do violations count for each agent separately or do they occur and attribute to an agent group?
- Is the planning performed by a centralised mechanism, by one of the agents and then distributed or by each one of the agents?
- By what protocols do agents communicate their preferences over the plans? How are these preferences weighed against other agents' preferences?
- By what criteria can the overall cost of the agents' plans be evaluated when dealing with many agents? Are social criteria implicated in this evaluation or do we only measure the agent's welfare?
- What is the expected overhead for multiple agent planning and how affordable is it when dealing with real time communities?

6.3.2 Probabilistic Practical Normative Reasoning

In order for our approach to be even closer to real life situations, we have started investigating the extension of our norm semantics to include a probabilistic observation of violations of norms. We consider this a more realistic situation, since it is not always the case that a norm violation is noted and registered, therefore, many times the consequences of violations and the need to apply sanctions are nonexistent. A draft for the implementation of such a framework could be the following.

In the specification of a norm, we introduce a probability that the system "observes" an infraction that might occur for every norm (that could for example added to the activating condition of a repair norm, as a predicate *Bernoulli(p)* that would at every

point in time be true with a possibility p). Additionally, a slightly more advanced model of the world descriptions, where actions have stochastic consequences might be adopted to make the domains more realistic. Dynamic Bayesian Networks, which are extended Bayesian networks (BDNs) [Ben-Gal, 2007] (graphical models that encode probabilistic relationships among variables of interest) can be used for this purpose.

In order to implement the probabilistic normative planner, we choose the newly introduced language RDDL [Sanner, 2010]. RDDL (Relational Dynamic influence Diagram Language) was introduced at the International Probabilistic Planning Competition (IPPC)[1] and its semantics is based on Dynamic Bayesian Networks and support basic probability distributions and stochastic predicates. An objective function specifies how the immediate rewards should be optimised over time for optimal control. We would assume that agents operate in fully observed environment. For such cases, RDDL is just a factored Markov Decision Process (MDP)[2], so in principle, RDDL aims to support the representation and simulation of a wide spectrum of relational MDPs. The core problem in probabilistic planning is, given an MDP, to find a policy for the decision maker (planner), i.e. a function that specifies the action that the decision maker will choose when in a state. Note that once a Markov decision process is combined with a policy in this way, this fixes the action for each state and the resulting combination behaves like a Markov chain.

In order to simulate the MDP, a software tool acting as the environment throughout the MDP is needed. At the IPPC'2011, the rddlsim[3] software was presented. rddlsim implements a parser and *client* (*probabilistic planner*)/*server* simulation architecture for RDDL. Nevertheless, since rddlsim does not represent a multi-agent environment, we can modify and extend 2APL to play the same role as the server part of rddlsim, that is, to act as the environment, keeping at the same time the rest of the framework intact.

The process roughly is integrated into the agent's deliberation cycle as follows[4]:

1. The initial state of a problem is described in RDDL and loaded both by the server (2APL) and the client (probabilistic planner represented by the rddlsim client).
2. The agent's goals are incorporated into the RDDL problem, with the state where they are reached having very high weight

[1] http://users.cecs.anu.edu.au/ ssanner/IPPC_2011/

[2] MDP is a discrete time stochastic control process. At each time step, the process is in some state s, and the decision maker may choose any action α that is available in state s'. The process responds at the next time step by randomly moving into a new state , and giving the decision maker a corresponding reward $R_a(s,s')$. The objective is to find *policies* that maximise the expected sum of rewards accumulated over a (potentially infinite) horizon. That is, to find a policy π that gives the optimal action $\pi(s) \in A$ for each state. The Markov property means that the transition probabilities $P_a(s,s')$ only depend on the current state s and the action a (so no information about past states is used).

[3] http://code.google.com/p/rddlsim/

[4] We only show the parts that concern the planning/next action picking process, assuming the rest such as update of beliefs, update of the goal base etc. remain the same.

3. The client at every cycle chooses an action it wants to execute. It does so by simulating lots of states internally until some timeout is reached. Then, it submits the action believes to be best to the server.
4. The server gets the action, and applies it to the current state. As we are in a probabilistic setting, the outcome of applying the action to the real world state is a non-deterministic successor state (the next real world state). That state is returned to the planner, and the whole procedure goes back to step 2.
5. If the finite horizon is reached or the goals are achieved or modified the run terminates.

Such an extension of 2APL, would result in a multi-agent probabilistic environment where the agents' actions as well as the norms do not have deterministic consequences, but instead, the agent's deliberation is done by taking into account the probability of getting "caught" for every possible infraction.

Publications of the Author

[PVSAN+08] Sofia Panagiotidi, Javier Vázquez-Salceda, Sergio Álvarez-Napagao, Sandra Ortega-Martorell, Steven Willmott, Roberto Confalonieri and Patrick Storms, *Intelligent Contracting Agents Language*, In Proceedings of the Symposium on Behaviour Regulation in Multi-Agent Systems (BRMAS'08), Aberdeen, UK (2008).

[CANP+08] Roberto Confalonieri, Sergio Álvarez-Napagao, Sofia Panagiotidi, Javier Vázquez-Salceda and Steven Willmott, *A Middleware Architecture for Building Contract-Aware Agent-Based Services*, In Proceedings of the International Workshop on Service-Oriented Computing: Agents, Semantics and Engineering (SOCASE@AAMAS'08), Estoril, Portugal, ISBN 978-3-540-79967-2 (2008).

[OPVMLM+08] Nir Oren, Sofia Panagiotidi, Javier Vázquez-Salceda, Sanjay Modgil, Michael Luck and Simon Miles, *Towards a Formalisation of Electronic Contracting Environments*, In Proceedings of Coordination, Organizations, Institutions and Norms (COIN@AAAI'08), Chicago, Illinois (2008).

[VCGSKPA+09] Javier Vázquez-Salceda, Roberto Confalonieri, Ignasi Gómez-Sebastià, Patrick Storms, Nick Kuijpers, Sofia Panagiotidi and Sergio Álvarez, *Modelling Contractually-Bounded Interactions in the Car Insurance Domain*, In Proceedings of the First International ICST Conference on Digital Business (DIGIBIZ'09), London, (2009), ISBN: 978-963-9799-56-1.

[PNV+09] Sofia Panagiotidi, Juan Carlos Nieves and Javier Vázquez-Salceda, *A Framework to Model Norm Dynamics in Answer Set Programming*, In Proceedings of the Workshop on Formal Approaches to Multi-Agent Systems (FAMAS'09), Torino, Italy (2009).

[AGPTOV+11] Sergio Álvarez-Napagao, Ignasi Gómez-Sebastià, Sofia Panagiotidi, Arturo Tejeda, Luis Oliva and Javier Vázquez-Salceda, *Socially-Aware Emergent Narrative*, In Proceedings of the Workshop on the uses of Agents for Educational Games and Simulations (AEGS@AAMAS'11), Taipei, Taiwan (2011).

[PV+11] Sofia Panagiotidi and Javier Vázquez-Salceda, *Towards Practical Normative Agents: a Framework and an Implementation for Norm-Aware Planning*, In Proceedings of the 13th International Workshop on Coordination, Organizations, Institutions and Norms in Agent Systems (COIN@WI-IAT'11), Lyon, France (2011).

[PVV+12] Sofia Panagiotidi, Javier Vázquez-Salceda and Wamberto Vasconcelos, *Contextual Norm-based Plan Evaluation via Answer Set Programming*, In Proceedings of Trust, Incentives and Norms in open Multi-Agent Systems Workshop at the 10th International Conference on Practical Applications of Agents and Multi-Agent Systems (TINMAS@PAAMS'12), Salamanca, Spain (2012).

[PVD+12] Sofia Panagiotidi, Javier Vázquez-Salceda and Frank Dignum, *Reasoning over Norm Compliance via Planning*, 13th International Workshop on Coordination, Organizations, Institutions and Norms in Agent Systems (COIN@AAMAS'12), Valencia, Spain (2012).

[PAV+13] Sofia Panagiotidi, Sergio Álvarez-Napagao and Javier Vázquez-Salceda, *Towards the Norm-Aware Agent: Bridging the Gap Between Deontic Specifications and Practical Mechanisms for Norm Monitoring and Norm-Aware Planning*, 15th International Workshop on Coordination, Organizations, Institutions and Norms in Agent Systems (COIN @AAMAS'13), St. Paul, Minnesota (2013).

Bibliography

Abrahams, A. S. and Bacon, J. M. (2002). The life and times of identified, situated, and conflicting norms. In *Proceedings of the 6th International Workshop on Deontic Logic in Computer Science (DEON'02)*, pages 3–20.

Adal, A. (2010). An interpreter for organization oriented programming language (2OPL). Master's thesis, Utrecht University.

Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., and Torroni, P. (2008). Verifiable agent interaction in abductive logic programming: the SCIFF framework. *ACM Transactions on Computational Logic (TOCL)*, 9(4):29.

Aldewereld, H. (2007). *Autonomy vs. Conformity: An Institutional Perspective on Norms and Protocols*. PhD thesis, Utrecht University.

Aldewereld, H., Álvarez-Napagao, S., Dignum, F., and Vázquez-Salceda, J. (2010). Making Norms Concrete. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'10)*.

Aldewereld, H. and Dignum, V. (2011). OperettA: Organization-oriented development environment. *Languages, Methodologies, and Development Tools for Multi-Agent Systems, Lecture Notes in Computer Science*, 6822:1–18.

Aldewereld, H., Grossi, D., Vázquez-Salceda, J., and Dignum, F. (2006). Designing Normative Behaviour Via Landmarks . *Coordination, Organizations, Institutions, and Norms in Multi-Agent Systems, Lecture Notes in Computer Science*, 3913:157–169.

Alechina, N., Dastani, M., and Logan, B. (2012). Programming norm-aware agents. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'12)*, pages 1057–1064. International Foundation for Autonomous Agents and Multiagent Systems.

Alechina, N., Dastani, M., and Logan, B. (2013). Reasoning about normative update. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI'13)*, pages 20–26.

Álvarez-Napagao, S., Aldewereld, H., Vázquez-Salceda, J., and Dignum, F. (2010). Normative monitoring: semantics and implementation. In *Proceedings of the 6th*

International Conference on Coordination, Organizations, Institutions, and Norms in Agent Systems (COIN@AAMAS'10). Springer-Verlag.

Ancona, D., Bailyn, L., and Brynjolfsson, E. (2003). What Do We Really Want? A Manifesto for the Organizations of the 21st Century. *MIT Discussion Paper*, pages 1–8.

Andrieux, A., Czajkowski, K., Dan, A., Keahey, K., Ludwig, H., Nakata, T., Pruyne, J., Rofrano, J., Tuecke, S., and Xu, M. (2007). Web Services Agreement Specification (WS-Agreement) Version 2005/09. Technical report.

Andrighetto, G. and Conte, R. (2012). Cognitive dynamics of norm compliance. From norm adoption to flexible automated conformity. *Artificial Intelligence and Law*, 20(4):359–381.

Antoniou, G., Dimaresis, N., and Governatori, G. (2008). A system for modal and deontic defeasible reasoning. In *Proceedings of the 2008 ACM Symposium on Applied computing (SAC'08)*, pages 2261–2265, New York, New York, USA. ACM Press.

Antoniou, G. and van Harmelen, F. (2003). Web Ontology Language: OWL. *Handbook on Ontologies in Information Systems*, pages 67–92.

Artikis, A. (2003). *Executable specification of open norm-governed computational systems*. PhD thesis, Imperial College London.

Artikis, A. and Sergot, M. (2010). Executable specification of open multi-agent systems. *Logic Journal of IGPL*, 18(1):31–65.

Artikis, A., Sergot, M., and Pitt, J. (2003). Specifying electronic societies with the Causal Calculator. *Agent-Oriented Software Engineering III, Lecture Notes in Computer Science*, 2585:1–15.

Artikis, A., Sergot, M., and Pitt, J. (2009). Specifying norm-governed computational societies. *ACM Transactions on Computational Logic (TOCL)*, 10(1):1–42.

Artosi, A., Cattabriga, P., and Governatori, G. (1994). KED: A Deontic Theorem Prover. *Proceedings of Legal Application of Logic Programming (ICLP'94)*, pages 60–76.

Aştefănoaei, L., Dastani, M., Meyer, J.-J., and de Boer, F. S. (2009). On the Semantics and Verification of Normative Multi-Agent Systems. *International Journal of Universal Computer Science*, 15(13):2629–2652.

Aştefănoaei, L., de Boer, F. S., and Dastani, M. (2010). The Refinement of Choreographed Multi-Agent Systems. In *Proceedings of the 7th International Conference on Declarative Agent Languages and Technologies (DALT'09)*, pages 20–34. Springer.

Bacchus, F. and Kabanza, F. (2000). Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1-2):123–191.

Bandara, A. K., Lupu, E. C., and Russo, A. (2003). Using Event Calculus to Formalise Policy Specification and Analysis. In *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'03)*, pages 26–39. IEEE Computer Society.

Baral, C. (2003). *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press.

Baral, C. and Gelfond, M. (1999). Reasoning agents in dynamic domains. In *Logic-based artificial intelligence, Kluwer Academic Publishers Norwell, MA, USA*, pages 257–279.

Baral, C., Gelfond, M., and Provetti, A. (1997). Representing actions: Laws, observations and hypotheses. *The Journal of Logic Programming, Elsevier Science Inc, NY, USA*, 31(1-3):201–243.

Basin, D., Doser, J., and Lodderstedt, T. (2005). Model driven security. *Engineering Theories of Software Intensive Systems, NATO Science Series*, 195:353–398.

Bellifemine, F., Poggi, A., and Rimassa, G. (1999). JADE–A FIPA-compliant agent framework. 99(97-108).

Bellini, P., Mattolini, R., and Nesi, P. (2000). Temporal logics for real-time system specification. *ACM Computing Surveys (CSUR)*, 32(1):12–42.

Bellman, R. (1957). A Markovian decision process. *Journal of Mathematics and Mechanics*, 6:679–684.

Belnap, N. and Perloff, M. (1988). Seeing to it that: a canonical form for agentives. *Theoria*, 54(3):175–199.

Ben-Gal, I. (2007). Bayesian networks. In *Encyclopedia of Statistics in Quality & Reliability*. Wiley & Sons.

Bentzen, M. M. (2010). *Stit, Iit, and Deontic Logic for Action Types*. PhD thesis, Section for Philosophy and Science Studies, Roskilde University.

Billari, F. C. (2000). Social norms and life course events: A topic for simulation? *Workshop in Norms and Institutions in Multi-Agent Systems, ACM-AAAI*, pages 13–14.

Blum, A. L. and Furst, M. L. (1997). Fast planning through planning graph analysis. *Artificial intelligence, Elsevier Science BV*, (90):281–300.

Boella, G., Broersen, J., and van der Torre, L. W. N. (2008a). Reasoning about Constitutive Norms, Counts-As Conditionals, Institutions, Deadlines and Violations. In *Proceedings of the 11th Pacific Rim International Conference on Multi-Agents (PRIMA'08)*, pages 86–97, Hanoi, Vietnam.

Boella, G. and van der Torre, L. W. N. (2004). Regulative and constitutive norms in normative multiagent systems. In *Proceedings of the 9th International Conference on the Principles of Knowledge Representation and Reasoning (KR'04)*, pages 255–265.

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

Boella, G., van der Torre, L. W. N., and Verhagen, H. (2008b). Introduction to the special issue on normative multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 17(1):1–10.

Bonet, B., Loerincs, G., and Geffner, H. (1997). A robust and fast action selection mechanism for planning. In *Proceedings of the 14th National Conference on Artificial Intelligence and 9th Conference on Innovative Applications of Artificial Intelligence (AAAI'97/IAAI'97)*, pages 714–719.

Bordini, R. and Hübner, J. F. (2006). BDI agent programming in AgentSpeak using Jason. *Computational Logic in Multi-Agent Systems*, pages 143–164.

Bou, E., López-Sánchez, M., and Rodríguez-Aguilar, J.-A. (2007). Adaptation of autonomic electronic institutions through norms and institutional agents. *Engineering Societies in the Agents World VII, Lecture Notes in Computer Science*, 4457:300–319.

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

Bratman, M. E., Israel, D., and Pollack, M. (1991). Plans And Resource-Bounded Practical Reasoning. *In Robert Cummins and John L Pollock (eds), Philosophy and AI: Essays at the Interface, The MIT Press, Cambridge, Massachussets*, pages 1–22.

Braubach, L., Pokahr, A., and Lamersdorf, W. (2005). Jadex: A BDI-agent system combining middleware and reasoning. *Software Agent-Based Applications, Platforms and Development Kits, Birkhäuser-Verlag*, pages 143–168.

Broersen, J., Dastani, M., Hulstijn, J., Huang, Z., and van der Torre, L. W. N. (2001). The BOID architecture: conflicts between beliefs, obligations, intentions and desires. In *Proceedings of the 5th International Conference on Autonomous Agents (AGENTS'01)*, pages 9–16.

Brunel, J., Bodeveix, J.-P., and Filali, M. (2006). A state/event temporal deontic logic. *Deontic Logic and Artificial Normative Systems, Lecture Notes in Computer Science*, 4048:85–100.

Cardoso, H. L. and Oliveira, E. (2009). A context-based institutional normative environment. *Coordination, Organizations, Institutions and Norms in Agent Systems IV, Lecture Notes in Computer Science*, 5428:140–155.

Cardoso, H. L. and Oliveira, E. (2010). Directed deadline obligations in agent-based business contracts. *Coordination, Organizations, Institutions and Norms in Agent Systems V, Lecture Notes in Computer Science*, 6069:225–240.

Carmo, J. and Pacheco, O. (2000). Deontic and action logics for collective agency and roles. In *Proceedings of the 5th International Workshop on Deontic Logic in Computer Science (DEON'00)*, pages 93–124.

Casali, A. (2008). *On intentional and social agents with graded attitudes*. PhD thesis, Universitat de Girona, Spain.

Castelfranchi, C., Dignum, F., Jonker, C. M., and Treur, J. (2000). Deliberative Normative Agents: Principles and Architecture. *Intelligent Agents VI. Agent Theories, Architectures, and Languages Lecture Notes in Computer Science*, 1757:364–378.

Cliffe, O. (2007). *Specifying and analysing institutions in multi-agent systems using answer set programming*. PhD thesis, Department of Computer Science, University of Bath.

Cliffe, O., de Vos, M., and Padget, J. (2007a). Answer set programming for representing and reasoning about virtual institutions. *Computational Logic in Multi-Agent Systems, Lecture Notes in Computer Science, Springer-Verlag*, 4371:60–79.

Cliffe, O., de Vos, M., and Padget, J. (2007b). Specifying and reasoning about multiple institutions. *Coordination, Organizations, Institutions, and Norms in Agent Systems II, Lecture Notes in Computer Science*, 4386:67–85.

Cohen, P. R. and Levesque, H. (1990). Intention is choice with commitment. *Artificial Intelligence, Elsevier Science Publishers BV (North-Holland)*, (42):213–261.

Conte, R. and Castelfranchi, C. (2001). Are Incentives Good Enough To Achieve (Info) Social Order? *Social Order in Multiagent Systems, Springer*.

Cranefield, S. and Winikoff, M. (2011). Verifying social expectations by model checking truncated paths. *Journal of Logic Computation*, 21(6):1217–1256.

Cranefield, S., Winikoff, M., and Vasconcelos, W. (2012). Modelling and monitoring interdependent expectations. *Coordination, Organizations, Institutions, and Norms in Agent System VII, Lecture Notes in Computer Science*, 7254:149–166.

Craven, R. and Sergot, M. (2008). Agent strands in the action language nC+. *Journal of Applied Logic*, 6(2):172–191.

Criado, N., Argente, E., and Botti, V. (2010a). A BDI architecture for normative decision making. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'10)*, pages 1383–1384. International Foundation for Autonomous Agents and Multiagent Systems.

Criado, N., Argente, E., Noriega, P., and Botti, V. (2010b). Towards a Normative BDI Architecture for Norm Compliance. In *Proceedings of the 11th International Workshop on Coordination, Organization, Institutions and Norms in Multi-Agent Systems (COIN@MALLOW'10)*, pages 65–81, Lyon, France.

da Silva Figueiredo, K., da Silva, V. T., and de Oliveira Braga, C. (2011). Modeling norms in multi-agent systems with NormML. *Coordination, Organizations, Institutions, and Norms in Agent Systems VI, Lecture Notes in Computer Science*, 6541:39–57.

Daskalopulu, A. (2000). Modelling Legal Contracts as Processes. In *Proceedings of the 11th International Workshop on Database and Expert Systems Applications (DEXA'00)*, pages 1074–1079.

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

Dastani, M., Dignum, V., and Dignum, F. (2003). Role-assignment in open agent societies. In *Proceedings of the 2nd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'03)*, pages 489–496.

Dastani, M., Grossi, D., Meyer, J.-J., and Tinnemeier, N. (2009a). Normative multi-agent programs and their logics. *Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany*.

Dastani, M., Tinnemeier, N., and Meyer, J.-J. (2009b). A Programming Language for Normative Multi-Agent Systems. *In Information Science Reference, Hershey, PA, USA*.

Davidsson, P. (2000). Emergent Societies of Information Agents. *Coordination, Organizations, Institutions and Norms in Agent Systems V, Lecture Notes in Computer Science*, 1860:143–153.

Davidsson, P. (2001). Categories of Artificial Societies. *Engineering Societies in the Agents World II, Lecture Notes in Computer Science*, 2203:1–9.

Davis, R. and King, J. (1975). An overview of production systems. *Machine Intelligence 8: Machine Representations of Knowledge, Wiley*, pages 300–331.

de Barros Paes, R., de Cerqueira Gatti, M. A. A., de Carvalho, G. R., Rodrigues, L. F., and de Lucena, C. J. (2006). A Middleware for Governance in Open Multi-Agent Systems. *Technical Report 33/06, Pontifical Catholic University of Rio de Janeiro*.

De Silva, L., Sardina, S., and Padgham, L. (2009). First principles planning in BDI systems. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'09)*, pages 1105–1112. International Foundation for Autonomous Agents and Multiagent Systems.

Demolombe, R. and Louis, V. (2006). Speech acts with institutional effects in agent societies. *Deontic Logic and Artificial Normative Systems, Lecture Notes in Computer Science*, 4048:101–114.

Dennett, D. C. (1989). *The intentional stance*. MIT Press.

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

Dignum, F., Broersen, J., Dignum, V., and Meyer, J.-J. (2005). Meeting the Deadline: Why, When and How. *Formal Approaches to Agent-Based Systems, Lecture Notes in Computer Science*, 3228:30–40.

Dignum, F., Morley, D., Sonenberg, E. A., and Cavedon, L. (2000). Towards socially sophisticated BDI agents. In *Proceedings of the 4th International Conference on MultiAgent Systems (ICMAS'00)*, pages 111–118.

Dignum, V. (2004). *A Model for Organizational Interaction: Based on Agents, Founded in Logic*. PhD thesis, Universiteit Utrecht.

Dignum, V. (2009). The Role of Organization in Agent Systems. *In V Dignum, editor, Handbook of Research on Multi-Agent Systems: Semantics and Dynamics of Organizational Models Information Science Reference, IGI Global*.

Dignum, V., Meyer, J.-J., Dignum, F., and Weigand, H. (2002). Formal specification of interaction in agent societies. *In 2nd Goddard Workshop on Formal Approaches to Agent-Based Systems (FAABS), Springer*, pages 37–52.

Dignum, V., Vázquez-Salceda, J., and Dignum, F. (2004). A model of almost everything: Norms, structure and ontologies in agent organizations. In *Proceedings of the 3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'04)*, pages 1498–1499. IEEE Computer Society.

Dimopoulos, Y., Nebel, B., and Koehler, J. (1997). Encoding planning problems in nonmonotonic logic programs. *Recent Advances in AI Planning, Lecture Notes in Computer Science*, 1348:169–181.

Dybalova, D., Testerink, B., Dastani, M., Logan, B., Dignum, F., and Chopra, A. (2013). A Framework for Programming Norm-Aware Multi-Agent Systems. In *Proceedings of the 15th International Workshop on Coordination, Organizations, Institutions, and Norm in Multi-Agent Systems (COIN@AAMAS'13)*, Minneapolis, USA.

Edelkamp, S. and Hoffmann, J. (2004). PDDL2. 2: The Language for the Classical Part of the 4th International Planning Competition. Technical report.

Eiter, T., Faber, W., Leone, N., Pfeifer, G., and Polleres, A. (2003). A logic programming approach to knowledge-state planning, II: the DLVk system. *Artificial intelligence, Elsevier Science BV*, 144(1-2):157–211.

Eshghi, K. (1988). Abductive Planning with Event Calculus. In *Proceedings of the Fifth International Conference on Logic Programming*, pages 562–579.

Esteva, M. (2003). *Electronic Institutions: from specification to development*. PhD thesis, Institut d'Investigació en Intel.ligència Artificial (IIIA), Universitat Autonòma de Barcelona.

Esteva, M., Padget, J., and Sierra, C. (2002). Formalizing a language for institutions and norms. *Intelligent Agents VIII, Lecture Notes in Computer Science*, 2333:348–366.

Esteva, M. and Sierra, C. (2002). Islander 1.0 language definition. Technical Report IIIA-TR-02-02, 2002.

Fagundes, M. S., Billhardt, H., and Ossowski, S. (2010). Reasoning about Norm Compliance with Rational Agents. In *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI'10)*, pages 1027–1028.

Fagundes, M. S., Ossowski, S., Luck, M., and Miles, S. (2012a). Representing and Evaluating Electronic Contracts with Normative Markov Decision Processes. *AI Communications*, 25:1–17.

Fagundes, M. S., Ossowski, S., Luck, M., and Miles, S. (2012b). Using normative markov decision processes for evaluating electronic contracts. *AI Communications*, 25(1):1–17.

Farrell, A. D. H., Sergot, M., Sallé, M., and Bartolini, C. (2005). Using the event calculus for tracking the normative state of contracts. *International Journal of Cooperative Information Systems, World Scientific Publishing Company*.

Ferber, J., Gutknecht, O., and Michel, F. (2004). From agents to organizations: an organizational view of multi-agent systems. *Agent-Oriented Software Engineering IV, Lecture Notes in Computer Science*, 2935:214–230.

Fikes, R. E. and Nilsson, N. J. (1972). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3):189–208.

Finin, T., Labrou, Y., and Mayfield, J. (1995). KQML as an agent communication language. *MIT Press*.

Fisher, M. (1997). A Normal Form for Temporal Logics and its Applications in Theorem-Proving and Execution. *Journal of Logic and Computation*, 7:429–456.

Fisher, M. (2008). Temporal Representation and Reasoning. *Handbook of Knowledge Representation, Elsevier B.V.*

Fornara, N. and Colombetti, M. (2009). Specifying and Enforcing Norms in Artificial Institutions. *Declarative Agent Languages and Technologies VI, Lecture Notes in Computer Science*, 5397:1–17.

Fox, M. and Long, D. (2009). PDDL 2.1 : An Extension to PDDL for Expressing Temporal Planning Domains. Technical report.

García-Camino, A., Rodríguez-Aguilar, J.-A., Sierra, C., and Vasconcelos, W. (2006). A rule-based approach to norm-oriented programming of electronic institutions. *ACM SIGecom Exchanges*, 5(5):33–40.

García-Camino, A., Rodríguez-Aguilar, J.-A., Sierra, C., and Vasconcelos, W. (2009). Constraint rule-based programming of norms for electronic institutions. *Autonomous Agents and Multi-Agent Systems*, 18(1).

Gasser, L. (1992). An Overview of DAI. *Distributed Artificial Intelligence: Theory and Praxis, Kluwer Academic Publishers*, pages 9–30.

Gâteau, B., Boissier, O., Khadraoui, D., and Dubois, E. (2005). MoiseInst: An Organizational Model for Specifying Rights and Duties of Autonomous Agents. In *Proceedings of the 3rd European Workshop on Multi-Agent Systems (EUMAS'05)*, pages 484–485. Citeseer.

Gelfond, M. and Lifschitz, V. (1993). Representing Action and Change by Logic Programs. *Journal of logic programming*.

Gelfond, M. and Lifschitz, V. (1998). Action Languages. *Electronic Transactions on AI*, (3).

Gelfond, M. and Lobo, J. (2008). Authorization and obligation policies in dynamic systems. *Logic Programming, Lecture Notes in Computer Science*, 5366:22–36.

Georgeff, M. P. and Lansky, A. L. (1987). Reactive reasoning and planning. In *Proceedings of the 6th National Conference in AI (AAAI'87)*, pages 677–682.

Gerevini, A. and Long, D. (2006). Plan constraints and preferences in PDDL3: The Language of the Fifth International Planning Competition. Technical report.

Ghallab, M., Howe, A., Knoblock, C., Mcdermott, D., Ram, A., Veloso, M., Weld, D. S., and Wilkins, D. (1998). PDDL—the planning domain definition language Version 1.2. *AIPS98 planning committee*, 78(4):1–27.

Ghallab, M., Nau, D. S., and Traverso, P. (2004). *Automated Planning: Theory & Practice*. Morgan Kaufmann, 1 edition.

Giunchiglia, E., Lee, J., Lifschitz, V., McCain, N., and Turner, H. (2004). Nonmonotonic Causal Theories. *Artificial Intelligence*, 153(1-2):49–104.

Gödel, K. (1931). Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38(1):173–198.

Gómez-Sebastia, I., Álvarez-Napagao, S., Vázquez-Salceda, J., and Felipe, L. O. (2012). Towards Runtime Support for Norm Change from a Monitoring Perspective. In *Proceedings of the 1st International Conference on Agreement Technologies (AT'12)*, pages 71–85.

Governatori, G. (2005). Representing business contracts in RuleML. *International Journal of Cooperative Information Systems*, 14:181–216.

Governatori, G., Gelati, J., Rotolo, A., and Sartor, G. (2002). Actions, institutions, powers: preliminary notes. In *Proceedings of the 1st International Workshop on Regulated Agent-Based Social Systems: Theories and Applications (RASTA'02)*, pages 131–147.

Governatori, G. and Rotolo, A. (2010). Norm compliance in business process modeling. In *Semantic Web Rules, Lecture Notes in Computer Science*, pages 194–209. Springer-Verlag.

Grossi, D. (2007). *Designing invisible handcuffs: Formal investigations in institutions and organizations for multi-agent systems*. PhD thesis, Utrecht University.

Grossi, D. and Dignum, F. (2005). From abstract to concrete norms in agent institutions. *Formal Approaches to Agent-Based Systems*, pages 12–29.

Hannoun, M., Boissier, O., Sichman, J. S., and Sayettat, C. (2000). MOISE: An organizational model for multi-agent systems. *Springer*, pages 156–165.

Hansen, J., Pigozzi, G., and van der Torre, L. W. N. (2007). Ten philosophical problems in deontic logic. In Boella, G., van der Torre, L. W. N., and Verhagen, H., editors, *Normative Multi-agent Systems, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany*.

Hilpinen, R., editor (1971). *Deontic Logic: Introductory and Systematic Readings*. Synthese Library. Springer.

Hindin, M. J. (2007). *Role Theory*. Blackwell Encyclopedia of Sociology, Oxford, UK, Malden, USA and Carlton, Australia.

Hindriks, K. V. (2009). Programming Rational Agents in GOAL. *Multi-Agent Programming: Languages, Tools and Applications*, pages 119–157.

Hindriks, K. V., van der Hoek, W., and van Riemsdijk, M. B. (2009). Agent programming with temporally extended goals. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'09)*, pages 137–144. International Foundation for Autonomous Agents and Multiagent Systems.

Hindriks, K. V. and van Riemsdijk, M. B. (2013). A Real-Time Semantics for Norms with Deadlines. In *Proceedings of the International Conference on Autonomous Agents and MultiAgent Systems (AAMAS'13)*, pages 1–8.

Hodgson, G. M. (2006). What Are Institutions? *Journal of Economic Issues 2006*, vol. 40(1):2–4.

Hoffmann, J. and Nebel, B. (2001). The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*.

Hohfeld, W. N. (1917). Fundemental Legal Conceptions as Applied in Judicial Reasoning. *Yale Law Journal*, pages 711–770.

Horn, A. (1951). On sentences which are true of direct unions of algebras. *The Journal of Symbolic Logic*, 16(1):14–21.

Horty, J. F. (2001). *Agency and Deontic Logic* . Oxford University Press.

Hsu, C.-W. and Wah, B. W. (2008). The sgplan planning system in ipc-6. *Sixth International Planning Competition, Sydney, Australia (September 2008)*.

Hübner, J. F., Boissier, O., and Bordini, R. (2011). A normative programming language for multi-agent organisations. *Annals of Mathematics and Artificial Intelligence*, 62(1-2):27–53.

Hübner, J. F., Sichman, J. S., and Boissier, O. (2002). MOISE+: towards a structural, functional, and deontic model for MAS organization. In *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'02)*, pages 501–502.

Jones, A. J. I., Artikis, A., and Pitt, J. (2013). The design of intelligent socio-technical systems. *Artificial Intelligence Review*, 39(1):5–20.

Jones, A. J. I. and Sergot, M. (1993). On the characterisation of law and computer systems: the normative systems perspective. *Deontic logic in computer science: normative system specification*, pages 275–307.

Jones, A. J. I. and Sergot, M. (1996). A formal characterisation of institutionalised power. *Logic Journal of IGPL*, 4(3):427.

Jung, C. G., Fischer, K., and Burt, A. (1996). Multi-agent planning using an abductive: event calculus. *Technical Report DFKI, DFKI Research Reports*, 96-04:114.

Kanger, S. (1972). Law and logic. *Theoria*, 38(3):105–132.

Kanger, S. and Stenlund, S. (1974). *Logical theory and semantic analysis: essays dedicated to Stig Kanger on his fiftieth birthday*. Springer.

Kautz, H. and Selman, B. (1992). Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI'92)*, pages 359–363.

Kollingbaum, M. (2005). *Norm-governed practical reasoning agents*. PhD thesis, University of Aberdeen.

Kollingbaum, M., Jureta, I. J., Vasconcelos, W., and Sycara, K. (2008). Automated Requirements-Driven Definition of Norms for the Regulation of Behavior in Multi-Agent Systems. In *Proceedings of the Artificial Intelligence and the Simulation of Behaviour Convention in Multi-Agent Systems (AISB'08)*.

Kollingbaum, M. and Norman, T. J. (2003). Norm adoption in the NoA agent architecture. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'03)*, pages 1038–1039.

Kotok, A. and Webber, D. (2001). ebXML: The New Global Standard. *New Riders Publishing*, page 339.

Kowalski, R. A. and Sergot, M. (1986). A logic-based calculus of events. *New Generation Computing*, 4(1):67–95.

Koymans, R. (1990). Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299.

Lam, J., Guerin, F., Vasconcelos, W., and Norman, T. J. (2010). Building multi-agent systems for workflow enactment and exception handling. In *Coordination, Organizations, Institutions and Norms in Agent Systems V, Lecture Notes in Computer Science*, pages 53–69. Springer-Verlag.

Lam, J., Vasconcelos, W., Guerin, F., Corsar, D., Chorley, A., Norman, T. J., Vázquez-Salceda, J., Panagiotidi, S., Confalonieri, R., and Gómez-Sebastia, I. (2009). ALIVE: A framework for flexible and adaptive service coordination. In *Proceedings of the 10th International Workshop on Engineering Societies in the Agents World X (ESAW'09)*, pages 236–239. Springer.

Leff, L. and Meyer, P. (2007). OASIS LegalXML eContracts Version 1.0 Committee Specification 1.0, 27 April 2007.

Levesque, H., Pirri, F., and Reiter, R. (1998). Foundations for the situation calculus. *Linköping Electronic Articles in Computer and Information Science*, 3.

Likhachev, M., Ferguson, D., Gordon, G., Stentz, A., and Thrun, S. (2005). Anytime Dynamic A*: An Anytime, Replanning Algorithm. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS'05), American Association for Artificial Intelligence*.

Liu, C. L. and Layland, J. W. (1973). Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61.

Lokhorst, G.-J. C. (1999). Ernst Mally's Deontik (1926). *Notre Dame Journal of Formal Logic*, 40(2):273–282.

Lomuscio, A., Qu, H., and Raimondi, F. (2009). MCMAS: A Model Checker for the Verification of Multi-Agent Systems. *Computer Aided Verification, Lecture Notes in Computer Science*, 5643:682–688.

Lomuscio, A., Qu, H., and Solanki, M. (2011). Towards Verifying Contract Regulated Service Composition. *Journal of Autonomous Agents and Multi-Agent Systems*, 24(3):345–373.

López y López, F. (2003). *Social Power and Norms: Impact on agent behaviour*. PhD thesis, University of Southampton.

López y López, F. and Luck, M. (2002). Towards a Model of the Dynamics of Normative Multi-Agent Systems. In *Proceedings of the International Workshop on Regulated Agent-Based Social Systems: Theories and Applications (RASTA '02)*, pages 175–194.

López y López, F., Luck, M., and d'Inverno, M. (2001). A Framework for Norm-Based Inter-Agent Dependence. In *Proceedings of the 3rd Mexican International Conference on Computer Science, SMCC-INEGI*, pages 31–40.

López y López, F., Luck, M., and d'Inverno, M. (2004). Normative agent reasoning in dynamic societies. In *Proceedings of the 3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'04)*, pages 732–739.

López y López, F., Luck, M., and d'Inverno, M. (2006). A normative framework for agent-based systems. *Computational & Mathematical Organizational Theory, Springer Science + Business Media, LLC*, (12):227–250.

Ludwig, H., Keller, A., Dan, A., King, R. P., and Franck, R. (2003). Web Service Level Agreement (WSLA) Language Specification. Technical report.

Mally, E. (1926). Grundgesetze des Sollens: Elemente der Logik des Willens. *Graz: Leuschner und Lubensky, Universitäts-Buchhandlung*.

Marek, V. W. and Truszczynski, M. (1999). Stable models and an alternative logic programming paradigm. *The Logic Programming Paradigm, Artificial Intelligence*, pages 375–398.

McCain, N. C. (1997). Causality in Commonsense Reasoning about Actions. *The University of Texas at Austin*.

McCarthy, J. and Hayes, P. J. (1969). Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502.

Meneguzzi, F. and Luck, M. (2008). Composing high-level plans for declarative agent programming. *Declarative Agent Languages and Technologies V Lecture Notes in Computer Science*, 4897:69–85.

Meneguzzi, F. and Luck, M. (2009a). Leveraging new plans in AgentSpeak (PL). *Declarative Agent Languages and Technologies VI, Lecture Notes in Computer Science*, 5397:111–127.

Meneguzzi, F. and Luck, M. (2009b). Norm-based behaviour modification in BDI agents. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'09)*. International Foundation for Autonomous Agents and Multiagent Systems.

Meneguzzi, F., Vasconcelos, W., and Oren, N. (2010). Using constraints for Norm-aware BDI Agents. In *Proceedings of the 4th Annual Conference of the International Technology Alliance*, London, UK.

Meneguzzi, F., Vasconcelos, W., Oren, N., and Luck, M. (2012). Nu-BDI: Norm-aware BDI Agents. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'09)*.

Milosevic, Z. and Dromey, G. R. (2002). On Expressing and Monitoring Behaviour in Contracts. In *Proceedings of the 6th International Enterprise Distributed Object Computing Conference (EDOC'02)*, pages 3–14.

Milosevic, Z., Jøsang, A., Dimitrakos, T., and Patton, M. A. (2002). Discretionary Enforcement of Electronic Contracts. In *Proceedings of the 6th International Enterprise Distributed Object Computing Conference (EDOC'02)*, pages 39–50. IEEE Computer Society.

Modgil, S., Faci, N., Meneguzzi, F., Oren, N., Miles, S., and Luck, M. (2009). A framework for monitoring agent-based normative systems. *International Foundation for Autonomous Agents and Multiagent Systems*, 1:153–160.

Moses, Y. and Tennenholtz, M. (1995). Artificial Social Systems. *Computers and Artificial Intelligence*, 14(6):533–562.

Nau, D. S., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D., and Yaman, F. (2003). SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research*, pages 379–404.

Noriega, P. (1997). *Agent Mediated Auctions: The Fishmarket Metaphor*. PhD thesis, Universitat Autonoma de Barcelona.

Noriega, P. and Sierra, C. (2002). *Electronic Institutions: Future Trends and Challenges*. Springer-Verlag.

North, D. C. (1990). *Institutions, institutional change, and economic performance*. Cambridge University Press.

Odell, J. J., van Dyke Parunak, H., and Fleischer, M. (2003). The Role of Roles in Designing Effective Agent Organizations. *Software Engineering for Large-Scale Multi-Agent Systems, Lecture Notes in Computer Science*, 2603:27–38.

Oh, J., Meneguzzi, F., and Sycara, K. (2011). Prognostic agent assistance for norm-compliant coalition planning. In *Proceedings of the 2011 Conference on Autonomous Agents and Multi-Agent Systems*, Taipei, Taiwan.

Okouya, D. and Dignum, V. (2008). OperettA: a prototype tool for the design, analysis and development of multi-agent organizations. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'08)*, pages 1677–1678. International Foundation for Autonomous Agents and Multiagent Systems.

Oren, N., Luck, M., and Miles, S. (2010). A model of normative power. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'10)*, pages 815–822. International Foundation for Autonomous Agents and Multiagent Systems.

Oren, N. and Meneguzzi, F. (2013). Norm Identification through Plan Recognition. In *Proceedings of the 15th International Workshop on Coordination, Organisations, Institutions and Norms (COIN@AAMAS'13)*, Saint Paul, MN, USA.

Oren, N., Panagiotidi, S., Vázquez-Salceda, J., Modgil, S., Luck, M., and Miles, S. (2009). Towards a Formalisation of Electronic Contracting Environments. *Coordination, Organizations, Institutions and Norms in Agent Systems IV, Lecture Notes in Computer Science*, 5428:156–171.

Oren, N., Vasconcelos, W., Meneguzzi, F., and Luck, M. (2011). Acting on norm constrained plans. *Computational Logic in Multi-Agent Systems, Lecture Notes in Computer Science*, 6814:347–363.

Ostrom, E. (1986). An agenda for the study of institutions. *Public Choice*, 48(1):3–25.

Padgham, L. and Winikoff, M. (2004). *Developing intelligent agent systems: a practical guide*. John Wiley and Sons.

Panagiotidi, S., Álvarez-Napagao, S., and Vázquez-Salceda, J. (2013). Towards the Norm-Aware Agent: Bridging the gap between deontic specifications and practical mechanisms for Norm Monitoring and Norm-Aware Planning. In *Proceedings o the 15th International Workshop on Coordination, Organizations, Institutions and Norms (COIN@AAMAS'13)*, Minneapolis, USA.

Panagiotidi, S., Nieves, J. C., and Vázquez-Salceda, J. (2009). A framework to model norm dynamics in Answer Set Programming. In *Proceedings of the Workshop on Formal Approaches to Multi-Agent Systems (FAMAS'09)*, pages 193–201.

Panagiotidi, S., Vázquez-Salceda, J., Ortega-Martorell, S., Jakob, M., Solanki, M., Álvarez-Napagao, S., Oren, N., Confalonieri, R., Biba, J., and Willmott, S. (2008). Contracting Language Syntax and Semantics Specifications. Technical report.

Paschke, A. (2005). RBSLA: A declarative Rule-based Service Level Agreement Language based on RuleML. In *Proceedings of the International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce (CIMCA'05)*, pages 308–314.

Paschke, A., Dietrich, J., and Kuhla, K. (2005). A Logic Based SLA Management Framework. *Semantic Web and Policy Workshop (SWPW), 4th Semantic Web Conference (ISWC 2005), Galway, Ireland*.

Pednault, E. P. D. (1994). ADL and the State-Transition Model of Action. *Journal of Logic and Computation*, 4(5):467–512.

Penner, J. (1988). The Rules of Law: Wittgenstein, Davidson, and Weinrib's Formalism. *University of Toronto Faculty of Law Review*, 46.

Pörn, I. (1974). Some basic concepts of action. *In Stig Kanger, Sören Stenlund (eds): Logical theory and semantic analysis: essays dedicated to Stig Kanger on his fiftieth birthday, Volume 63 of Synthese Dordrecht / Library, Springer*.

Prakken, H. and Sergot, M. (1996). Contrary-to-duty obligations. *Studia Logica*, 57(1):91–115.

Prakken, H. and Sergot, M. (1997). Dyadic deontic logic and contrary-to-duty obligations. *Defeasible Deontic Logic, Synthese Library*, 263:223–262.

Puterman, M. L. (1994). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Series in Probability and Statistics. John Wiley & Sons, Inc.

Ranathunga, S., Cranefield, S., and Purvis, M. (2012). Integrating expectation monitoring into BDI agents. *Programming Multi-Agent Systems, Lecture Notes in Computer Science*, 7217:74–91.

Rao, A. S. (1996). AgentSpeak (L): BDI agents speak out in a logical computable language. In *Proceedings of the 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96)*, pages 42–55, Eindhoven, The Netherlands. Springer.

Rao, A. S. and Georgeff, M. P. (1995). BDI Agents: From Theory to Practice. In *Proceedings of the 1st International Conference on Multi-Agent Systems (ICMAS'95)*, pages 312–319.

Sanner, S. (2010). Relational dynamic influence diagram language (RDDL): Language description. Technical report.

Santos, F., Jones, A. J. I., and Carmo, J. M. C. L. M. (1997). Action concepts for describing organised interaction. In *Proceedings of the 30th Hawaii International Conference on System Sciences*, pages 373–382. IEEE Comput. Soc. Press.

Sardina, S., De Silva, L., and Padgham, L. (2006). Hierarchical planning in BDI agent programming languages: A formal approach. In *Proceedings of the 5th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'06)*, pages 1001–1008. ACM.

Scott, W. R. (1995). *Institutions and organizations*. Sage Publications.

Searle, J. R. (1969). Speech acts: An essay in the philosophy of language. *Cambridge University Press*.

Searle, J. R. (1997). The construction of social reality. page 256.

Sergot, M. (2003). (C+)++: An action language for modelling norms and institutions. *Technical Report, Imperial College*.

Sergot, M. and Craven, R. (2006). The Deontic Component of Action Language nC+. *Deontic Logic and Artificial Normative Systems, Lecture Notes in Computer Science*, 4048:222–237.

Shanahan, M. (2000). An abductive event calculus planner. *The Journal of Logic Programming*, 44(1):207–240.

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

Sierra, C., Garcia, P., and Arcos, J. L. (2001). On the formal specification of Electronic Institutions. *Agent Mediated Electronic Commerce, Lecture Notes in Computer Science*, 1991:126–147.

Sierra, C., Rodríguez-Aguilar, J.-A., Noriega, P., Esteva, M., and Arcos, J. L. (2004). Engineering multi-agent systems as electronic institutions. *European Journal for the Informatics Professional*, 4(4):33–39.

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

Tauriainen, H. (2006). *Automata and linear temporal logic: translations with transition-based acceptance*. PhD thesis, Helsinki University of Technology.

The JBoss Drools team (2013). *Drools Introduction and General User Guide*, 5.4.0.cr1 edition.

Thiébaux, S., Hoffmann, J., and Nebel, B. (2005). In defense of PDDL axioms. *Artificial Intelligence*, 168(1):38–69.

Thielscher, M. (1999). From situation calculus to fluent calculus: State update axioms as a solution to the inferential frame problem. *Artificial Intelligence*, 111(1–2):277–299.

Thielscher, M. (2005). FLUX: A logic programming method for reasoning agents. *Theory and Practice of Logic Programming*, 5(4-5).

Tufis, M. and Ganascia, J.-G. (2012). Normative rational agents-A BDI approach. In *Proceedings of the 1st workshop on Rights and Duties of Autonomous Agents, European Conference on Artificial Intelligence (RDA2@ECAI'12)*, page 38.

van der Torre, L. W. N. (2003). Contextual deontic logic: Normative agents, violations and independence. *Annals of Mathematics and Artificial Intelligence*, 37(1-2):33–63.

van Riemsdijk, M. B., Dennis, L. A., Fisher, M., and Hindriks, K. V. (2013). Agent reasoning for norm compliance: a semantic approach. In *Proceedings of the 12th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'13)*, pages 499–506. International Foundation for Autonomous Agents and Multiagent Systems.

van Steen, M., Pierre, G., and Voulgaris, S. (2012). Challenges in very large distributed systems. *Journal of Internet Services and Applications*, 3(1):59–66.

Vanhee, L., Aldewereld, H., and Dignum, F. (2011). Implementing Norms? In *Proceedings of the 2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology (WI-IAT'11)*, pages 13–16, Lyon, France.

Vasconcelos, W., Kollingbaum, M., and Norman, T. J. (2007). Resolving conflict and inconsistency in norm-regulated virtual organizations. In *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'07)*.

Vasconcelos, W., Kollingbaum, M., and Norman, T. J. (2009). Normative conflict resolution in multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 19(2):124–152.

Vázquez-Salceda, J. (2004). The role of Norms and Electronic Institutions in Multi-Agent Systems. *Whitestein Series in Software Agent Technology, Birkhäuser Verlag AG, Switzerland, ISBN 3-7643-7057-2*.

Vázquez-Salceda, J. and Dignum, F. (2003). Modelling electronic organizations. *Multi-Agent Systems and Applications III, Lecture Notes in Computer Science*, 2691:584–593.

Vázquez-Salceda, J., Dignum, V., and Dignum, F. (2005). Organizing multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 11(3):307–360.

Vecht, B. v. d. (2009). Adjustable Autonomy: Controling Influences on Decision Making. *Utrecht University, SIKS Dissertation Series*.

Villatoro, D., Andrighetto, G., Sabater-Mir, J., and Conte, R. (2011). Dynamic Sanctioning for Robust and Cost-Efficient Norm Compliance. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI'11)*, pages 414–419.

von Wright, G. H. (1951). Deontic Logic. *Mind*, 60(237):1–15.

von Wright, G. H. (1956). A note on deontic logic and derived obligation. *Mind*, 65:507–509.

von Wright, G. H. (1971). A New System of Deontic Logic. *Deontic Logic: Introductory and Systematic Readings, Springer Netherlands*, 33:105–120.

Wallace, R. J. (2009). Practical Reason. *The Stanford Encyclopedia of Philosophy, http://plato.stanford.edu/entries/practical-reason/*.

Walter, R. (1996). Jorgensen's Dilemma and How to Face It. *Ratio Juris*, 9(2):168–171.

Wang, P.-H. (2010). Are Rules Exclusionary Reasons in Legal Reasoning? *Archiv für rechts-und sozialphilosophie, ARSP. Beiheft*, (119):37–48.

Wang, Y. and Ruhe, G. (2007). The cognitive process of decision making. *International Journal of Cognitive Informatics and Natural Intelligence (IJCINI)*, 1(2):73–85.

Weiss, G. (1999). *Multiagent systems: a modern approach to distributed artificial intelligence*. The MIT Press.

Weld, D. S. (1999). Recent advances in AI planning. *AI magazine*, 20(2):93–123.

Winikoff, M. (2005). JACK^{TM} intelligent agents: An industrial strength platform. *Multi-Agent Programming, Multiagent Systems, Artificial Societies, and Simulated Organizations*, 15:175–193.

Winikoff, M., Padgham, L., Harland, J., and Thangarajah, J. (2002). Declarative and procedural goals in intelligent agent systems. In *Proceedings of the 8th International Conference on Principles of Knowledge Representation and Reasoning (KR'02)*, pages 470–481. Morgan Kaufman.

Wooldridge, M. J. (2001). *An introduction to multiagent systems*. John Wiley & Sons.

Wooldridge, M. J. and Jennings, N. R. (1995). Intelligent Agents: Theory and Practice. *Knowledge engineering review*, 10:115–152.

Wooldridge, M. J., Jennings, N. R., and Kinny, D. (2000). The Gaia Methodology for Agent-Oriented Analysis and Design. *Autonomous Agents and Multi-Agent Systems, Kluwer Academic Publishers, The Netherlands*, (3):285–312.

Appendix A

Basis Framework Semantics

This Appendix presents an extract of our work on formalising normative elements and norm modelling [Oren et al., 2009] previous to this thesis. Although the norm semantics presented in Chapter 4 and 5 are based on different logical foundations, some concepts such as the basic steps on the norm cycle are inspired by this previous work.

At their core, normative environments impose a set of (possibly conditional) requirements on an agent's behaviour. These requirements may range from actions that the agent may, or should, undertake, to states of affairs within the environment that an agent may, should, or should not, allow to occur. To formalise a norm-aware language, one must also formalise its normative components. As will be discussed in Section A.5, researchers have provided many such formalisations, often in the context of deontic logic. Our interest in norms is more focused; as part of a norm-aware language, we are interested in tracking their changing state (for example, when they are "active", as well as the more traditional "violated"). Furthermore, our application domain requires slightly different philosophical assumptions when compared to those made in the deontic tradition, as we assume that norms can be violated, but may then, in some cases, be "un-violated".

A.1 Norms for Modelling Regulative Clauses

A normative environment is made up of various descriptive elements, for example, stating which ontologies may be used to explain the terms found within it. Most importantly, it specifies a set of *clauses*, each of which represents a *norm*.

Norms can be interpreted as socially derived prescriptions specifying that some set of agents (the norm's *targets*) may, or must, perform some action, or see that some state of affairs occurs. Norms can be understood as regulating the behaviour of agents. This is their role when encoded in normative environments.

Norms are social constructs, and we believe that it is meaningless to consider norms independently of their social aspect. This is because a norm is imposed on the target by some other entity (the imposer) which must be granted, via the society, some power to impose the norm. Without this power, the norm's target is free to ignore the norm's prescriptions. With the presence of power, a penalty may be imposed on an agent violating a norm. These penalties take on the form of additional norms, giving certain agents within a society the option to impose penalties.

When designing our normative model, we attempted to meet the following requirements, imposed upon us by the domain in which we operate:

- There should be a strong focus on semantics at all levels. The knowledge might be represented by ontologies and supported by semantic languages. The actions can be described by expressive languages and provide the option to be interpreted and used by different frameworks. The norms might be modelled using extended deontic logic.
- The framework should allow for the contextualisation of norms as well as the domain within which the agent operates
- The model should allow for the monitoring of norms. That is, it should allow for the determination of whether a violation took place and, if possible, who was responsible for causing the violation.
- Verification of norms should also be supported, i.e. determining whether conflicts between norms could occur, or whether a norm could never, sometimes, or always be complied with.
- Norms should be able to cope with contrary to duty obligations as well as conditions based on the status of other norms. For example, consider the pair of norms "One is obliged to park legally", and "If one parks illegally, one is obliged to pay a fine". The second norm carries normative weight only if the first norm is violated. These types of norms commonly appear within normative environments, and it is thus critical that our model is able to represent them.
- Norms must be able to cope with contrary to duty obligations, as well as conditions based on the status of other norms.
- The model should be extensible, allowing different knowledge representations and reasoning mechanisms to make use of it.

No requirement was placed on detecting and resolving normative conflict. Many such techniques exist, each having a different view of what constitutes normative conflict (e.g. [Vasconcelos et al., 2007]). It is intended that these techniques could make use of our framework for their underlying representation of norms. Similarly, our model should not prescribe what must occur when a violation is detected. Instead, we assume that the environment would contain clauses dealing with such situations.

Since it is possible that norms have a normative force only under specific situations, an *activation condition* is associated with them. In this way, norms remain *abstract* until

their activation condition becomes true, which is when they get *instantiated*. Once it gets instantiated, a norm stays active, regardless of its activation condition, until a particular *expiration condition* becomes true. When this happens, the norm is assumed to no longer have normative force. Finally, in addition to these two conditions, the norm's *normative goal* is used to indicate when the norm gets violated. As stated in Section 3.2.6, obligations and prohibitions are the two *norm types* on which our framework focuses. Like others, we assume that additional norm types may be constructed from these basic types (e.g. a prohibition could be seen as an obligation with a negated normative goal).

Norms may be activated, met and discharged based on a number of factors including the status of other norms and the state of the environment (and the actions performed by other agents therein).

A.2 Formal Preliminaries

In the next sections, we formalise our notions of norms. We do so in a number of steps: first, we define their structure; after this is done, we show how the status of a norm may change over time. Before examining norms, we must define a number of related concepts.

We assume the use of a predicate based first-order language $\mathcal{L}$ containing logical symbols: connectives $\{\neg, \wedge, \vee, \rightarrow\}$, quantifiers $\{\forall, \exists\}$, an infinite set of variables, and the non-logical predicate, constant and function symbols. The standard definitions for free and bound variables, as well as ground formulas are assumed. Finally, the set of well formed formulas of $\mathcal{L}$ are denoted as $\mathit{wff}(\mathcal{L})$. A single well-formed formula from this set is denoted *wff*.

We make use of the standard notions of substitution of variables in a wff, where $V = \{x_1 \leftarrow t_1, x_2 \leftarrow t_2, ..., x_i \leftarrow t_i\}$ is a substitution of the terms $t_1, \ldots, t_n$ for variables $x_1, \ldots, x_n$ in a wff. If no variables exist in a *wff* resulting from a substitution, it is said to be fully grounded, and is partially grounded otherwise.

Our model allows us to infer predicates based on the status of the environment, clauses, and norms. We assume that other predicates may exist whose truth value may be inferred from other sources such as ontologies, or an action model. Each of these sources thus generates a theory, denoted by Γ. For example, we label the theory generated by the environment as Γ_{Env}. We label the union of all theories as Ω.

Formally, a *normative environment* contains a set of *clauses* representing *norms* imposed on *agents*. A normative environment that has been agreed to by those agents has normative force, and the agents affected by the environment's norms are the parties. Since a normative environment may be instantiated more than once with different agents playing similar roles, agents are identified using an indirection mechanism: a

normative environment imposes norms on a set of *roles*, and agents are associated with these roles before the environment is created.

A.3 Structural Definitions

We may now define the structure of norms and normative environment. Since these concepts act upon agents, we begin by defining these entities, as well as roles, which are names referenced to identify the agent upon which a norm acts.

A.3.1 Agent Names and Roles

Agents in our framework are left unspecified; we only assume that they are associated with a unique agent name[1].

A role may have one or more parent roles. This means that whenever an agent is assigned to a role, it is also assigned to that role's parent roles, and so assumes the clauses applying to those parents. If a role r_1 is a parent of role r_2, then r_2 may be referred to as a child role of r_1.

Definition A.1. (Roles) A role is a constant. We assume that the set of all roles is called *Roles*. Then a role hierarchy definition $RoleHierarchyDefinition$ is a binary relation of the form $(Parent, Child)$ where $Parent, Child \in Roles$.

If we would like to specify that the role of a repairer exists in the environment, and also include the fact that any car repairer also acts as a repairer (i.e. *repairer* is a parent role of *car repairer*), then $(repairer, carRepairer)$ would be contained within $RoleHierarchyDefinition$.

A.3.2 Norms

An environment contains a set of clauses, represented by norms. Norms may bind an agent to a certain course of action in all situations, or may only affect an agent when certain activation conditions apply. Similarly, once an agent achieves a certain state of affairs, a norm may no longer apply. Finally, norms affect only a specific set of target agents. A norm thus consists of the following.

- A type identifier, stating whether the norm is an obligation or a prohibition.
- An activation condition stating when the norm must be instantiated.

[1] We have chosen to consider all agents as black boxes and not impose restrictions on the agents' internal architecture. This is to make our framework as technology independent as possible and also to make our framework suitable to model Multi-agent setups mixing software agents and human agents.

- A normative goal or state (condition) used to identify when the norm is violated (in the case of both obligations and prohibitions).
- An expiration condition used to determine when the norm no longer affects the agent.
- A target, identifying either the agents or the agent roles which the norm affects.

Norms may be activated, met, and discharged based on various factors including the environment, and the status of other norms. We assume the existence of Ω, a theory (or possibly a set of theories) allowing one to interpret the status of norms[2]. To represent the status of a norm, we define a normative environment theory Γ_{NEnv} below, and assume that it is part of Ω.

A norm that may apply to a number of situations is, in a sense, abstract. When a situation to which it applies does arise, the norm is instantiated and exerts a normative force on the agents that are beholden to it. We may thus informally define an abstract norm as a norm that, when the conditions are right, comes into effect (i.e. is instantiated) and only then has normative force over one or more agents. As the name suggests, an instantiated norm is an instantiated abstract norm, which has normative power over a set of agents, until it is discharged.

A group of abstract norms (which, in our case, are the clauses of the environment) is gathered into an abstract norm store. Norms may be represented as a tuple of *wffs*.

Definition A.2. (Abstract Norms and Abstract Norm Store) An *Abstract Norm Store*, denoted $\mathcal{ANS}$, consists of a set of abstract norms, each of which is a tuple of the form

$$\langle NormType, NormActivation, NormCondition, NormExpiration, NormTarget\rangle$$

where:

- $NormType \in \{\text{obligation, prohibition}\}$
- $NormTarget \in Roles$
- for $N \in \{NormActivation, NormCondition, NormExpiration, NormTarget\}$, N is a *wff* (denoted by ϕ_N)

We may further divide *NormCondition* into a state-based maintenance condition (labeled *SMaintenanceCondition*) and an action-based maintenance condition (labeled *AMaintenanceCondition*). A truth value for *NormCondition* may be computed as the truth value of (*AMaintenanceCondition* $\wedge$ *SMaintenanceCondition*).

NormActivation is some *wff* ϕ_{NA} which, when entailed by the theory, must be entailed as the fully grounded ϕ'_{NA} in order that the abstract norm can be instantiated and thus come into force. The substitution of variables V such that $\phi'_{NA} = S(\phi_{NA})$ is then

[2] For example, Ω may include references to the environment, an ontology, an action model, and normative environment.

applied to the other components of the abstract norm, thus specifying the instantiated norm.

A.3.3 Instantiating Abstract Norms

We now define how abstract norms are instantiated with respect to the domain environment theory and normative environment theory.

An instantiated norm has the same overall form as an abstract norm but its activation condition is grounded, its remaining parameters are partially grounded using the same grounding as the activation condition and its *NormTarget* no longer refers to roles but to agents.

Definition A.3. (Instantiation of Abstract Norms)
The abstract norm

$$\langle NormType, NormActivation, NormCondition, NormExpiration, NormTarget \rangle$$

instantiated by the *Environment* Γ_{Env} and *Normative Environment Theory* Γ_{NEnv}, obtains an instantiated norm:

$$\langle NormType, NormActivation', NormCondition', NormExpiration', NormTarget' \rangle$$

where:

- $\Omega \vdash NormActivation'$, where $NormActivation'$ is fully grounded such that $NormActivation'$ = $S(NormActivation)$
- $NormCondition' = S(NormCondition)$
- $NormExpiration' = S(NormExpiration)$.
- $NormTarget' = \{X \mid \Omega \cup \{NormActivation'\} \cup \{S(NormTarget)\} \vdash X\}$, where $NormTarget' \subseteq AgentNames$

Notice that $NormTarget'$ is the set of individuals X to whom the instantiated norm applies. These individuals are identified with reference to (entailed by) the domain environment theory, normative environment[3], and the $NormActivation$ and $NormTarget$ *wffs* that are grounded with respect to the former environments. In the context of a clause, the norm's targets are identified by using the $RoleHierarchyDefinition$ relation. Note also that $NormCondition'$ an $NormExpiration'$ may only be partially grounded.

[3]The normative environment may be used when a target should be identified based on the status of another norm. For example, in the case of a contrary to duty obligation, a penalty must be paid by the agent(s) violating some other norm.

Given a set of abstract norms ANS, together with a Ω, we define the set of norms that may be instantiated from Ω as $inst(ANS)$.

A.4 Dynamic Semantics

So far, we have described the structure of the environment and norms. To meet the goals of monitoring and verification, the changing status of norms must be tracked. Usually, we will be interested on how the norms affect the parties, i.e. how, given the evolution of the environment, various norms are instantiated, violated and discharged. Additionally, agents, and other norms, may need to determine the current, or historic state of norms, allowing them, for example, to identify all the obligations and prohibitions they are committed to fulfil at any point in time. To do so, we define the **normative environment**, a structure which may be used to identify the status of norms as they go through their lifecycle, and show how it may be used to perform evaluations regarding the status of a norm.

To do this, we now describe the normative environment theory Γ_{NEnv}. This structure defines predicates that may be used to identify the status of norms as they progress through their lifecycle. A normative environment theory is built around a normative environment, which is itself a (possibly infinite) sequence of normative states $NS_1, NS_2, \ldots$, each of which identifies the norms whose status has changed at the point in time associated with the normative state. Each normative state NS_i in the sequence is defined with respect to the overarching theory Ω (which includes Γ_{NEnv}), and a given set of abstract norms ANS. Thus, a normative state represents an instance during which one or more events related to some norms' status occurred. Each normative state keeps track of four basic events, namely:

1. when an abstract norm is instantiated;
2. when an instantiated norm expires;
3. when a norm's normative condition holds;
4. when a norm's normative condition does not hold.

In order to formally define a normative state we first define the evaluation of an instantiated norm's *NormCondition* and *ExpirationCondition*:

Definition A.4. (The *holds()* Predicate) Let *in* be an instantiated norm

$$\langle NormType, NormActivation, NormCondition, NormExpiration, NormTarget\rangle$$

Then, for $N \in \{NormCondition,\ NormExpiration\}$:

- $holds(in, N)$ evaluates to true if $\Omega \vdash N'$, where N' is entailed with all variables in N grounded
- $holds(in, N)$ evaluates to *false* otherwise

Our formal definition of a normative state then identifies those instantiated norms whose normative condition evaluates to true, those whose normative condition evaluates to false, and those whose expiration condition evaluates to true:

Definition A.5. (Normative State) Let *INS* be a set of instantiated norms. A normative state NS, defined with respect to a set *INS* of instantiated norms, and domain environment theory Γ_{Env} and normative environment theory Γ_{NEnv}, is a tuple of the form: $\langle NSTrue, NSFalse, NSExpires \rangle$ where:

- $NSTrue = \{in \in INS |\ holds(in, NormCondition)$ is true$\}$
- $NSFalse = \{in \in INS |\ holds(in, NormCondition)$ is false$\}$
- $NSExpires = \{in \in INS |\ holds(in, NormExpiration)$ is true$\}$

Since $NSTrue \cup NSFalse \supseteq NSExpires$, it is sufficient to identify the instantiated norms in a normative state, denoted $inst_norms(NS)$, by the union of those norms whose normative condition evaluates to true, and those, whose normative condition evaluates to false. That is to say:

$$inst_norms(NS) = NSTrue \cup NSFalse$$

Definition A.6. (Normative Environment) A normative environment NE is a possibly infinite ordered sequence $NS_1, NS_2, \ldots$ where for $i = 1 \ldots$, we say that NS_i is the normative state *previous* to NS_{i+1}.

Given a normative state, the subsequent normative state is defined by removal of the expired instantiated norms, addition of new instantiated norms, and checking the norm state of all instantiated norms. We therefore define a minimal set of conditions that a normative environment should satisfy:

Definition A.7. (Normative State Semantics) Let $\mathcal{ANS}$ be an abstract norm store, NE the normative environment $NS_1, NS_2, \ldots$, and for $i = 1 \ldots$, Ω_i a set of *wffs* denoting the domain environment associated with NS_i. For $i = 1 \ldots$, let us define the set of potential norms for NS_i as those that:

1. are instantiated in the previous state NS_{i-1} ($inst_norms(NS_{i-1})$)
2. those in the abstract norm store that are instantiated w.r.t. Ω_i (i.e. $inst(\mathcal{ANS})$ as defined in Definition A.3).

and not those that have expired in the previous state, i.e. $NSExpires_{i-1}$.
That is to say, the set of potential norms $PNorms_i$ is defined as follows:

$$PNorms_i = inst_norms(NS_{i-1}) \cup inst(\mathcal{ANS}) \setminus NSExpires_{i-1}$$

Then $NS_i = \langle NSTrue_i, NSFalse_i, NSExpires_i \rangle$ is defined (as in Definition A.5) with respect to the set $PNorms_i$, and theory Ω_i.

We define $NS_0 = \langle NSTrue_0, NSFalse_0, NSExpires_0 \rangle$ where $NSTrue_0 = \{\}$, $NSFalse_0 = \{\}$ and $NSExpires_0 = \{\}$

We suggest the following basic set of predicates entailed by Γ_{NEnv}, and in this way characterise how Γ_{NEnv} may be partially specified by the normative environment.[4] In the following definitions we assume a normative environment $\{NS_1, NS_2, \ldots\}$ where $NS_i = \langle NSTrue_i, NSFalse_i, NSExpires_i \rangle$, and $i > 0$. We make use of the *Gödelisation* operator $\ulcorner . \urcorner$ for naming normative states in the object level language. That is, $\ulcorner NS_i \urcorner$ names normative state NS_i and allows us to use it within *wffs*.

Definition A.8. (The *instantiated()* predicate) $\Gamma_{NEnv} \vdash instantiated(\ulcorner NS_i \urcorner, in)$ iff $in \in inst_norms(NS_i)$ and $(in \notin inst_norms(NS_{i-1}) \vee in \notin NSExpires_{i-1})$. We define by default $\Gamma_{NEnv} \nvdash instantiated(\ulcorner NS_0 \urcorner, in)$.

Intuitively, $instantiated(NS_i, in)$ holds if the norm in becomes instantiated in NS_i. That is, $instantiated(\ulcorner NS_i \urcorner, in)$ evaluates to true if norm in was instantiated in NS_i, and either was not instantiated in NS_{i-1} or expired in NS_{i-1} (and thus becomes instantiated again in NS_i).

Definition A.9. (The *expires()* predicate) $\Gamma_{NEnv} \vdash expires(\ulcorner NS_i \urcorner, in)$ iff $in \in NSExpires_i$. We also define $\Gamma_{NEnv} \nvdash expires(\ulcorner NS_0 \urcorner, in)$.

The $expires()$ predicate holds if an instantiated norm in expired within the current normative state.

Definition A.10. (The *active()* predicate) $\Gamma_{NEnv} \vdash active(\ulcorner NS_i \urcorner, in)$ if and only if $instantiated(\ulcorner NS_i \urcorner, in)$, or else $(in \in inst_norms(NS_{i-1}) \wedge in \notin NSExpires_{i-1})$. We also define $\Gamma_{NEnv} \nvdash active(\ulcorner NS_0 \urcorner, in)$.

$active(\ulcorner NS_i \urcorner, in)$ holds if a norm in is instantiated within normative state NS_i. This could be because it was instantiated within that state, or because it was instantiated earlier and has not yet expired.

Definition A.11. (The *becomesTrue()* predicate) $\Gamma_{NEnv} \vdash becomesTrue(\ulcorner NS_i \urcorner, in)$ iff $in \in NSTrue_i$ and, either $in \in NSFalse_{i-1}$, or $instantiated(\ulcorner NS_i \urcorner, in)$.

Intuitively, a norm in becomes true in NS_i if its normative condition evaluates to true, and either it was false in state NS_{i-1}, or if not, then in is instantiated in NS_i.

[4]In general, by stating requirement that some first-order theory Γ entail $\phi_1, \ldots, \phi_n$, we are effectively providing a partial specification of Γ. In semantic terms, any model for Γ is also model for $\phi_1, \ldots, \phi_n$

Definition A.12. (The *becomesFalse()* predicate)
$\Gamma_{NEnv} \vdash becomesFalse(\ulcorner NS_i \urcorner, in)$ iff $in \in NSFalse_i$ and, either $in \in NSTrue_{i-1}$ or $instantiated(\ulcorner NS_i \urcorner, in)$.

Here, $becomesFalse(\ldots)$ is similar to $becomesTrue(\ldots)$, dealing with falsehood rather than truth. The next two predicates check whether a norm is active and true, respectively false, in some normative state.

Definition A.13. (The *isTrue()* predicate)
$\Gamma_{NEnv} \vdash isTrue(\ulcorner NS_i \urcorner, in)$ if and only if $becomesTrue(\ulcorner NS_i \urcorner, in)$, or else, $active(\ulcorner NS_i \urcorner, in)$ and $in \in NSTrue_{i-1}$.

Definition A.14. (The *isFalse()* predicate)
$\Gamma_{NEnv} \vdash isFalse(\ulcorner NS_i \urcorner, in)$ if and only if $becomesFalse(\ulcorner NS_i \urcorner, in)$, or else, $active(\ulcorner NS_i \urcorner, in)$ and $in \in NSFalse_{i-1}$

Definition A.15. (Properties of Γ_{NEnv}) $\Gamma_{NEnv} \vdash \neg x$ iff $\Gamma_{NEnv} \not\vdash x$. This implies that:

- $\Gamma_{NEnv} \nvdash \perp$.
- $\neg$ is given a negation as failure semantics.

Apart from these low level predicates, we may define additional useful predicates. Some of these determine the status of a norm, while others allow access its operation.

Definition A.16. (Norm access predicates) Given a norm N with norm type $Type$, activation condition $NormActivation$, expiration condition $NormExpiration$, a norm target set $NormTarget$, a normative condition with a state component $SMaintenanceCondition$ and an action component $AMaintenanceCondition$, the following predicates (which may operate on both abstract and instantiated norms) may be defined:

$type(N,X) = true$ iff $X = Type$, and *false* otherwise.

$normActivation(N,X) = true$ iff *NormActivation* unifies with X, and *false* otherwise.

$normSCondition(N,X) = true$ iff *SMaintenanceCondition* unifies with X, and *false* otherwise.

$normACondition(N,X) = true$ iff *AMaintenanceCondition* unifies with X, and *false* otherwise.

$normExpiration(N,X) = true$ iff *NormExpiration* unifies with X, and *false* otherwise.

$normTarget(N,A) = true$ iff there is a unification between some element of *NormTarget* and A.

We may define the following predicates based on the normative environment. These predicates form a basis for our normative environment theory Γ_{NEnv}:

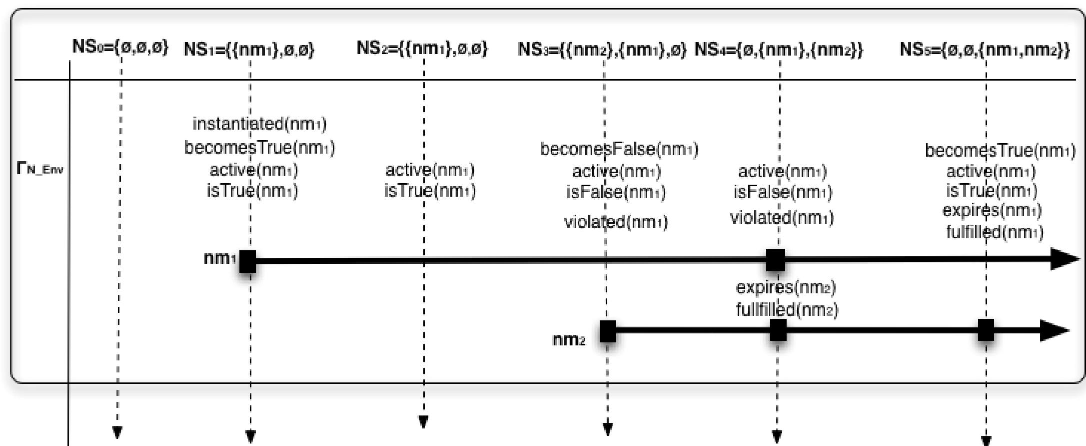


FIGURE A.1: Domain Environment and Normative Environment lifecycle

Definition A.17. (the violated() predicate)
$violated(\ulcorner NS_i \urcorner, in) = isFalse(\ulcorner NS_i \urcorner, in)$

The fulfilled predicate checks whether a norm has been fulfilled at a specific point in time

Definition A.18. (the fulfilled() predicate)
$fulfilled(\ulcorner NS_i \urcorner, in) = expires(\ulcorner NS_i \urcorner, in) \wedge \neg violated(\ulcorner NS_i \urcorner, in)$

$unfulfilled(\ulcorner NS_i \urcorner, in) = expires(\ulcorner NS_i \urcorner, in) \wedge violated(\ulcorner NS_i \urcorner, in)$

We may also be interested in determining whether a norm is a *violation handler*, that is, if it detects and handles the violation of some other clause. We make the simplifying assumption that a violation handler contains only the $violated()$ predicate in its activation condition.

Definition A.19. (the violationHandler() predicate)
$violationHandler(N) = normActivation(N, \ulcorner violated(X,Y) \urcorner)$ for any X, Y.

Finally, we may want to determine which norm (N_1) is the violation handler for another norm (N_2):

Definition A.20. (the handlesViolation() predicate)
$handlesViolation(N_1, N_2) = normActivation(N_1, \ulcorner violated(X, N_2) \urcorner)$ for any X

An example of how a normative theory evolves can be seen in Figure A.1.

A.5 Issues and Related Work

The normative framework we have described fulfils all of the requirements described in Section A.1. Not only are we able to detect whether a violation took place (via the

violation(...) predicate), but we may also detect the occurrence of additional *critical states* at which some normative event related state change took place. These critical states correspond to the various predicates described above. Additional, domain dependent critical states may be defined using the information found within the normative environment. Verification of a normative system may be performed by forward simulation over the domain and normative environments.

We assume that any norm-aware agent capable of being affected by norms, is associated with its own normative environment (and resulting normative environment theory). In fully observable environments, each agent's theory would be identical, but in other domains, these theories may diverge.

Our model does not describe what should occur if an obligation is violated. In Chapters 4 and 5 we assume that agents make use of an extension of this normative model to undertake their own practical reasoning. An agent may determine which norms affect it at any stage, and base its decisions on these.

One interesting aspect of our model (as illustrated by norm nm_1 in the example) is that norms may be violated for a certain period of time, after which they may return to an un-violated state. This is particularly useful as penalties may be assessed over the duration of a violation, with the norm still having normative force over an agent. This differs from the way most deontic theories deal with norms (for example [van der Torre, 2003]).

While a large variety of normative languages exist, many only specify an informal [Milosevic and Dromey, 2002], or programming language based [Kollingbaum, 2005] semantics. Formal languages, such as LCR [Dignum et al., 2002] have limited expressibility. Our approach of defining a rich language, and then constructing its semantics, is intended to overcome these weaknesses.

The work presented here has been inspired by a number of other researchers. For example, [Dignum, 2004] described the use of Landmarks as abstract states which "are defined as a set of propositions that are true in a state represented by the landmark". These landmarks may thus be seen as similar to critical states. The framework described by [Fornara and Colombetti, 2009] shares some similarities with our approach. Their focus on sanctions (which, in our model, are implemented more via additional norms) means that they only allow for very specific, predefined normative states, and that violations in their framework may only occur once.

Appendix B

Example Implementation Details

In this appendix the contextual layer for the example of Section 4.5 and the code for the implementations of Chapters 4 and 5 can be found.

B.1 Pizza Delivery Example Models

The metamodel of our conceptual framework of Section 3.2 as well as the models for the pizza delivery domain have been created using the Eclipse[1] developing framework. This section depicts screenshots taken from Eclipse's visual environment.

Figure B.1 depicts the top level element Normative Model.

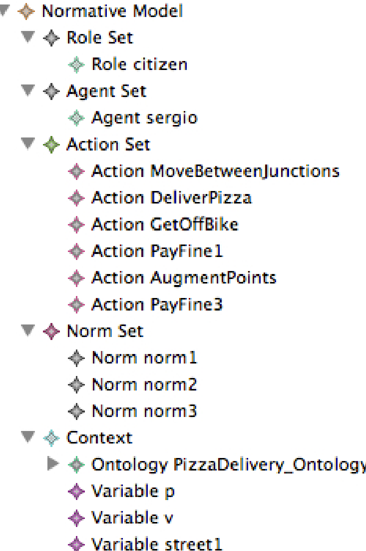


FIGURE B.1: Normative model representation

The various constants, variables and functions described in our example and used in the state formulas can be seen in Figure B.2.

[1]https://www.eclipse.org

Ontology PizzaDelivery_Ontology
Variable p
Variable v
Variable street1
Variable street2
Variable main
Variable sideStreet1
Variable sideStreet2
Variable someStreet
Variable speed
Constant sergio
Constant 0.5
Constant 1
Constant 2
Constant 3
Constant 10
Constant 20
Constant 30
Constant 40
Constant 60
Constant low
Constant medium
Constant high
Function vehicleSpeed(v)
Function fine(p)
Function penalty_points(p)
Function p_time
Function goalpizzatime(street1,street2)
Function speedLimit(street1)
Function penalty_points(sergio)
Function fine(sergio)
Function fun(1*((p_time/30)+1)+1*((penalty_points(sergio)/10)+1)+1*((fine(sergio)/60)+1))

FIGURE B.2: Variables, constants, functions representation

The ontology, containing the concepts used is depicted in Figure B.3.

Ontology PizzaDelivery_Ontology
Concept driving
Concept vehicle_at
Concept hasPizzaFor
Concept movedBetweenJunctions
Concept pizza_delivered
Concept connection
Concept fine_paid1
Concept fine_paid3
Concept points_augmented
Concept TRUE

FIGURE B.3: Ontology representation

The ontology concepts, combined with the different terms (variables, constants, functions) are used to define more complex formulas, such as atoms, negations, implications, etc. These state formulas (defining the conditions of the actions and the norms), can be seen in Figure B.4 (we only show part of these, since our model is bigger than what can fit here, so, we provide the reader a sample in order to get a pretty good idea of how they are formed).

- Atom driving(p,v)
- Negation -(driving(p,v))
- Atom vehicle_at(v,street1,street2)
- Negation -(vehicle_at(v,main,sideStreet1))
- Atom vehicle_at(v,street2,street1)
- Disjunction vehicle_at(v,street1,street2) or vehicle_at(v,street2,street1)
- Conjunction driving(p,v) and (vehicle_at(v,street1,street2) or (vehicle_at(street2,street1)) and hasPizzaFor(p,street1,street2)
- Atom pizza_delivered(p,street1,street2)
- Atom pizza_delivered(p,street2,street1)
- Atom hasPizzaFor(p,street1,street2)
- Atom hasPizzaFor(p,street2,street1)
- Negation -(hasPizzaFor(p,street1,street2))
- Negation -(hasPizzaFor(p,street2,street1))
- Relation Atom p_time += 0.5
- Relation Atom p_time += 1
- Relation Atom p_time += 2
- Relation Atom p_time += 3
- Relation Atom speedLimit(street1) leq vehicleSpeed(v)

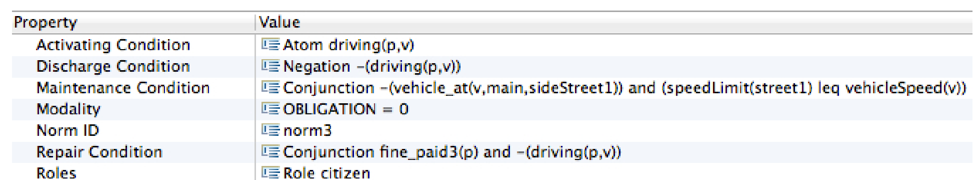
FIGURE B.4: Some of the state formulas used in the preconditions and effects of the actions and in the conditions of the norms

An example of the `DeliverPizza` action can be seen in Figure B.5.

Property	Value
Effect	Conjunction pizza_delivered(p,street1,street2) and pizza_delivered(p,street2,street1) and -(hasPizzaFor(p,street1,street2)) and -(hasPizzaFor(p,street2,street1)) and (p_time += 3)
Name	DeliverPizza
Parameters	Variable p, Variable street1, Variable street2
Precondition	Conjunction driving(p,v) and (vehicle_at(v,street1,street2) or (vehicle_at(street2,street1)) and hasPizzaFor(p,street1,street2)
Roles	Role citizen

FIGURE B.5: The action DeliverPizza representation

An example of **norm3** of the pizza delivery example can be seen in Figure B.6.



Property	Value
Activating Condition	Atom driving(p,v)
Discharge Condition	Negation -(driving(p,v))
Maintenance Condition	Conjunction -(vehicle_at(v,main,sideStreet1)) and (speedLimit(street1) leq vehicleSpeed(v))
Modality	OBLIGATION = 0
Norm ID	norm3
Repair Condition	Conjunction fine_paid3(p) and -(driving(p,v))
Roles	Role citizen

FIGURE B.6: norm3 representation

B.2 Pizza Delivery Example TLPLAN Code

```
1  (clear-world-symbols)

3  (set-search-strategy best-first)
   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
5  ;;; 1. The world symbols.
   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
7  (declare-described-symbols

9   (predicate connection 3)
    (predicate goalpizzaat 3)
11
    (predicate person 1)
```

```
13  (predicate street 1)
    (predicate vehicle 1)
15  (predicate speed_type 1)
    (predicate timeout 0)
17
    (predicate driving 2)
19  (predicate vehicle_at 3)
    (predicate movedBetweenJunctions 4)
21  (predicate pizza_delivered 3)
    (predicate hasPizzaFor 3)
23  (predicate fine_paid1 1)
    (predicate points_augmented 1)
25  (predicate fine_paid3 1)

27  (predicate true_predicate 0)
    (predicate false_predicate 0)
29
    (function p_time 0)
31  (function fine 1)
    (function penalty_points 1)
33  (function goalpizzatime 2)
    (function vehicleSpeed 1)
35  (function speedLimit 1)
    (function speedValue 1)
37 )

39 (declare-defined-symbols
           (function fun 0)
41         (predicate goalAchieved 0)

43         (predicate act-cond-norm1 2)
           (predicate maint-cond-norm1 1)
45         (predicate disch-cond-norm1 2)
           (predicate repair-cond-norm1 2)
47
           (predicate act-cond-norm2 3)
49         (predicate maint-cond-norm2 3)
           (predicate disch-cond-norm2 3)
51         (predicate repair-cond-norm2 1)

53         (predicate act-cond-norm3 2)
           (predicate maint-cond-norm3 1)
55         (predicate disch-cond-norm3 2)
           (predicate repair-cond-norm3 2)
57 )

59
   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
61 ;;;Operators
   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
63
   (def-adl-operator(MoveBetweenJunctions ?v ?main ?sideStreet1 ?sideStreet2 ?speed)
65  (pre
     (?v) (vehicle ?v)
67   (?main) (street ?main)
     (?sideStreet1) (street ?sideStreet1)
69   (?sideStreet2) (street ?sideStreet2)
     (?speed) (speed_type ?speed)
71   (and (or (vehicle_at ?v ?main ?sideStreet1) (vehicle_at ?v ?sideStreet1 ?main))
       (or (connection ?main ?sideStreet1 ?sideStreet2)
73         (connection ?main ?sideStreet2 ?sideStreet1))
     )
```

```
75  )
    (implies(vehicle_at ?v ?main ?sideStreet1)(del (vehicle_at ?v ?main ?sideStreet1)))
77  (implies(vehicle_at ?v ?sideStreet1 ?main)(del (vehicle_at ?v ?sideStreet1 ?main)))
    (add (vehicle_at ?v ?main ?sideStreet2))
79  (forall (?someStreet) (street ?someStreet)
      (implies (movedBetweenJunctions ?v ?main ?someStreet ?sideStreet1)
81      (del (movedBetweenJunctions ?v ?main ?someStreet ?sideStreet1))))
      (forall (?someStreet) (street ?someStreet)
83      (implies (movedBetweenJunctions ?v ?sideStreet1 ?someStreet ?main)
        (del (movedBetweenJunctions ?v ?sideStreet1 ?someStreet ?main))))
85  (add (movedBetweenJunctions ?v ?main ?sideStreet1 ?sideStreet2))
    (add (+= (p_time) (speedValue ?speed)))
87  ;;(implies (= ?speed low) (add (= (vehicleSpeed ?v) 20)))
    (implies (= ?speed medium) (add (= (vehicleSpeed ?v) 40)))
89  (implies (= ?speed high) (add (= (vehicleSpeed ?v) 60)))
    (cost  (+ (speedValue ?speed)))
91 )

93 (def-adl-operator (DeliverPizza ?p ?sideStreet1 ?sideStreet2)
    (pre
95   (?p) (person ?p)
     (?sideStreet1) (street ?sideStreet1)
97   (?sideStreet2) (street ?sideStreet2)
     (and (exists (?v) (driving ?p ?v)
99         (or (vehicle_at ?v ?sideStreet1 ?sideStreet2)
               (vehicle_at ?v ?sideStreet2 ?sideStreet1)))
101      (hasPizzaFor ?p ?sideStreet1 ?sideStreet2)
     )
103 )
    (add (pizza_delivered ?p ?sideStreet1 ?sideStreet2))
105 (add (pizza_delivered ?p ?sideStreet2 ?sideStreet1))
    (implies (hasPizzaFor ?p ?sideStreet1 ?sideStreet2)
107   (del (hasPizzaFor ?p ?sideStreet1 ?sideStreet2)))
    (implies (hasPizzaFor ?p ?sideStreet2 ?sideStreet1)
109   (del (hasPizzaFor ?p ?sideStreet2 ?sideStreet1)))
    (add (+= (p_time) 3))
111 (cost 3)

113 )

115 (def-adl-operator (PayFine1 ?p)
     (pre
117    (?p) (person ?p)
       (and (exists (?v) (vehicle ?v) (driving ?p ?v)) (not (fine_paid1 ?p)))
119  )
     (add (fine_paid1 ?p))
121  (add (+= (fine ?p) 30))
     (cost 30)
123 )

125 (def-adl-operator (AugmentPoints ?p)
     (pre
127    (?p) (person ?p)
       (not (points_augmented ?p))
129  )
     (add (points_augmented ?p))
131  (add (+= (penalty_points ?p) 10))
     (cost 10)
133 )

135 (def-adl-operator (PayFine3 ?p)
     (pre
```

```
    (?p) (person ?p)
    (and (exists (?v) (vehicle ?v) (driving ?p ?v)) (not (fine_paid3 ?p)))
  )
  (add (fine_paid3 ?p))
  (add (+= (fine ?p) 30))
  (cost 30)
)

(def-adl-operator (GetOffBike ?p ?v)
  (pre
    (?p) (person ?p)
    (?v) (vehicle ?v)
    (driving ?p ?v)
  )
  (del (driving ?p ?v))
  (add (+= (p_time) 0.5))
  (cost 0.5)
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;NORM 1
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(def-defined-predicate (act-cond-norm1 ?p ?v)
  (driving ?p ?v)
)
(def-defined-predicate (disch-cond-norm1 ?p ?v)
  (not (driving ?p ?v))
)
(def-defined-predicate (maint-cond-norm1 ?v)
  (not (exists (?main) (street ?main) (?Street1) (street ?Street1)
       (exists (?Street2) (street ?Street2)
        (and
          (movedBetweenJunctions ?v ?main ?Street1 ?Street2)
          (not (connection ?main ?Street1 ?Street2))
        )
  )))
)
(def-defined-predicate (repair-cond-norm1 ?p ?v)
  (and (fine_paid1 ?p) (not (driving ?p ?v)))
)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;NORM 2
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(def-defined-predicate (act-cond-norm2 ?p ?Street1 ?Street2)
  (and (or (hasPizzaFor ?p ?street1 ?street2)))
)
(def-defined-predicate (maint-cond-norm2 ?p ?Street1 ?Street2)
  (> (goalpizzatime ?Street1 ?Street2) (p_time))
)
(def-defined-predicate (disch-cond-norm2 ?p ?Street1 ?Street2)
  (or (pizza_delivered ?p ?Street1 ?Street2) )
)
(def-defined-predicate (repair-cond-norm2 ?p)
  (points_augmented ?p)
)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;NORM 3
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(def-defined-predicate (act-cond-norm3 ?p ?v)
  (driving ?p ?v)
)
(def-defined-predicate (maint-cond-norm3 ?v)
```

```
  (not (exists (?Street1) (street ?Street1)
       (exists (?Street2) (street ?Street2)
         (and (vehicle_at ?v ?Street1 ?Street2)
              (< (speedLimit ?Street1) (vehicleSpeed ?v))
         )
  )))
)
(def-defined-predicate (disch-cond-norm3 ?p ?v)
  (not (driving ?p ?v))
)

(def-defined-predicate (repair-cond-norm3 ?p ?v)
  (and (fine_paid3 ?p) (not (driving ?p ?v)))
)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;LTL CONTROL
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(set-tl-control

(forall (?p) (person ?p) (forall (?v) (vehicle ?v)
(forall (?street1) (street ?street1)
(forall (?street2) (street ?street2)
(and

(or
(always
  (not (act-cond-norm1 ?p ?v)))
(until
  (not (act-cond-norm1 ?p ?v))
  (and
    (act-cond-norm1 ?p ?v)
    (until (maint-cond-norm1 ?v) (disch-cond-norm1 ?p ?v))))
(until
  (not (act-cond-norm1 ?p ?v))
  (and
    (act-cond-norm1 ?p ?v)
    (until (maint-cond-norm1 ?v)
     (and
       (not (maint-cond-norm1 ?v))
       (until (not (timeout)) (repair-cond-norm1 ?p ?v))))))
)

(or
(always
  (not (act-cond-norm2 ?p ?street1 ?street2)))
(until
  (not (act-cond-norm2 ?p ?street1 ?street2))
  (and
    (act-cond-norm2 ?p ?street1 ?street2)
    (until (maint-cond-norm2 ?p ?street1 ?street2)
           (disch-cond-norm2 ?p ?street1 ?street2))))
(until
  (not (act-cond-norm2 ?p ?street1 ?street2))
  (and
    (act-cond-norm2 ?p ?street1 ?street2)
    (until (maint-cond-norm2 ?p ?street1 ?street2)
     (and
       (not (maint-cond-norm2 ?p ?street1 ?street2))
       (until (not (timeout)) (repair-cond-norm2 ?p))))))
)

(or
```

```
261 (always
      (not (act-cond-norm3 ?p ?v)))
263 (until
      (not (act-cond-norm3 ?p ?v))
265   (and
        (act-cond-norm3 ?p ?v)
267     (until (maint-cond-norm3 ?v) (disch-cond-norm3 ?p ?v))))
    (until
269   (not (act-cond-norm3 ?p ?v))
      (and
271     (act-cond-norm3 ?p ?v)
        (until (maint-cond-norm3 ?v)
273      (and
           (not (maint-cond-norm3 ?v))
275        (until (not (timeout)) (repair-cond-norm3 ?p ?v))))))
    )
277
    )))))
279 )
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
281 ;;;GOALS
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
283
    (def-defined-predicate (goalAchieved)
285   (forall (?p ?Street1 ?Street2) (goalpizzaat ?p ?Street1 ?Street2)
         (pizza_delivered ?p ?Street1 ?Street2))
287 )

289 (set-goal-addendum (goalAchieved))
```

Figure B.7: Pizza delivery example domain TLPLAN

```
 1 (define (problem escapeprobl1000) (:domain SofiaD)
   (:objects
 3         sergio - person
           bmw - vehicle
 5         paris
           corcega rosello provenca casanova villaroel muntaner urgell - street
 7         medium high - speed_type)
   (:init
 9
           (connection muntaner paris corcega)
11         (connection muntaner corcega rosello)
           (connection muntaner rosello provenca)
13
           (connection casanova corcega paris)
15         (connection casanova rosello corcega)
           (connection casanova provenca rosello)
17
           (connection villaroel paris corcega)
19         (connection villaroel corcega rosello)
           (connection villaroel rosello provenca)
21
           (connection urgell corcega paris)
23         (connection urgell rosello corcega)
           (connection urgell provenca rosello)
25
           (connection paris urgell villaroel)
27         (connection paris villaroel casanova)
```

```
        (connection paris casanova muntaner)

        (connection corcega muntaner casanova)
        (connection corcega casanova villaroel)
        (connection corcega villaroel urgell)

        (connection rosello urgell villaroel)
        (connection rosello casanova muntaner)

        (connection provenca muntaner casanova)
        (connection provenca casanova villaroel)
        (connection provenca villaroel urgell)

        (= (speedLimit paris) 50)
        (= (speedLimit corcega) 50)
        (= (speedLimit rosello) 50)
        (= (speedLimit provenca) 50)
        (= (speedLimit casanova) 50)
        (= (speedLimit villaroel) 50)
        (= (speedLimit muntaner) 50)
        (= (speedLimit urgell) 50)

        (= (speedValue high) 1)
        (= (speedValue medium) 1.5)
        ;;(= (speedValue low) 2)

        (hasPizzaFor sergio casanova corcega)
        (goalpizzaat sergio casanova corcega)
        (= (goalpizzatime casanova corcega) 21)

        (hasPizzaFor sergio urgell rosello)
        (goalpizzaat sergio urgell rosello)
        (= (goalpizzatime urgell rosello) 9)

        (vehicle_at bmw muntaner provenca)
        (movedBetweenJunctions bmw muntaner rosello provenca)
        (driving sergio bmw)
        (= (vehicleSpeed bmw) 10)

        (= (p_time) 0)
        (= (fine sergio) 0)
        (= (penalty_points sergio) 0)

        (true_predicate)
)

(:goal (and ))

(:metric minimize (fun))

)
```

FIGURE B.8: Pizza delivery example problem TLPLAN

B.3 Pizza Delivery Example PDDL Code

```
;; ../../../Metric-FF-v2.1/ff -o domain.pddl -f problem.pddl -w 3

(define (domain pizza_delivery)

(:predicates
  (speed_type ?speed)

  (driving ?p ?v)
  (vehicle_at ?v ?street ?street)
  (movedBetweenJunctions ?v -vehicle ?street ?street ?street)
  (connection ?main ?street1 ?street2)
  (pizza_delivered ?p ?street1 ?street2)
  (hasPizzaFor ?p ?street1 ?street2)
  (fine_paid1 ?p)
  (points_augmented ?p)
  (fine_paid3 ?p)

  (active_norm1 ?p -person ?v -vehicle)
  (inactive_norm1 ?p -person ?v -vehicle)
  (complied-with_norm1 ?p -person ?v -vehicle)
  (prev_active_norm1 ?p -person ?v -vehicle)
  (viol_norm1 ?p -person ?v -vehicle)
  (prev_viol_norm1 ?p -person ?v -vehicle)

  (active_norm1-rep ?p -person ?v -vehicle)
  (inactive_norm1-rep ?p -person ?v -vehicle)
  (prev_active_norm1-rep ?p -person ?v -vehicle)

  (active_norm2 ?p -person ?street1 ?street2)
  (inactive_norm2 ?p -person ?street1 ?street2)
  (complied-with_norm2 ?p -person ?street1 ?street2)
  (prev_active_norm2 ?p -person ?street1 ?street2)
  (viol_norm2 ?p -person ?street1 ?street2)
  (prev_viol_norm2 ?p -person ?street1 ?street2)

  (active_norm2-rep ?p ?street1 ?street2)
  (inactive_norm2-rep ?p ?street1 ?street2)
  (prev_active_norm2-rep ?p ?street1 ?street2)

  (active_norm3 ?p -person ?v -vehicle)
  (inactive_norm3 ?p -person ?v -vehicle)
  (complied-with_norm3 ?p -person ?v -vehicle)
  (prev_active_norm3 ?p -person ?v -vehicle)
  (viol_norm3 ?p -person ?v -vehicle)
  (prev_viol_norm3 ?p -person ?v -vehicle)

  (active_norm3-rep ?p -person ?v -vehicle)
  (inactive_norm3-rep ?p -person ?v -vehicle)
  (prev_active_norm3-rep ?p -person ?v -vehicle)

  (true)
)

(:functions (p_time)
            (fine ?p -person)
            (penalty_points ?p -person)
            (goalpizzatime ?street1 -street ?street2 -street)
            (vehicleSpeed ?v -vehicle)
            (speedLimit ?street -street)
```

```
                        (factor1))

;;;;;;;;;;;;;;;;; NORM 1;;;;;;;;;;;;;;;;;;;;;;;

(:derived (active_norm1 ?p -person ?v -vehicle)
  (driving ?p ?v))

(:derived (inactive_norm1 ?p -person ?v -vehicle)
  (not (driving ?p ?v)))

(:derived (viol_norm1 ?p -person ?v -vehicle)
  (and (active_norm1 ?p ?v)
       (exists (?main -street ?street1 -street ?street2 -street)
         (and (movedBetweenJunctions ?v ?main ?street1 ?street2)
              (not (connection ?main ?street1 ?street2))))))

(:derived (complied-with_norm1 ?p -person ?v -vehicle)
  (or
    (inactive_norm1 ?p ?v)
    (not (exists (?main -street ?street1 -street ?street2 -street)
           (and (movedBetweenJunctions ?v ?main ?street1 ?street2)
                (not (connection ?main ?street1 ?street2)))))))

(:derived (active_norm1-rep ?p -person ?v -vehicle)
  (and (or (and (viol_norm1 ?p ?v)
                (not (prev_viol_norm1 ?p ?v)))
           (prev_active_norm1-rep ?p ?v))
       (not (fine_paid1 ?p))))

(:derived (inactive_norm1-rep ?p -person ?v -vehicle)
  (or (and (or (or (not (driving ?p ?v))
                   (not (exists (?main -street ?street1 -street ?street2 -street)
                          (and (movedBetweenJunctions ?v ?main ?street1 ?street2)
                               (not (connection ?main ?street1 ?street2))))))
               (prev_viol_norm1 ?p ?v))
           (not (prev_active_norm1-rep ?p ?v)))
       (fine_paid1 ?p)))

;;;;;;;;;;;;;;;;; NORM 2;;;;;;;;;;;;;;;;;;;;;;;

(:derived (active_norm2 ?p -person ?street1 -street ?street2 -street)
  (and
    (or (hasPizzaFor ?p ?street1 ?street2) (prev_active_norm2 ?p ?street1 ?street2))
    (not (pizza_delivered ?p ?street1 ?street2))))

(:derived (inactive_norm2 ?p -person ?street1 -street ?street2 -street)
  (or
    (not (or (hasPizzaFor ?p ?street1 ?street2)
             (prev_active_norm2 ?p ?street1 ?street2)))
    (pizza_delivered ?p ?street1 ?street2)))

(:derived (viol_norm2 ?p -person ?street1 -street ?street2 -street)
  (and (active_norm2 ?p ?street1 ?street2)
       (and (hasPizzaFor ?p ?street1 ?street2)
            (> (p_time) (goalpizzatime ?street1 ?street2)))))

(:derived (complied-with_norm2 ?p -person ?street1 -street ?street2 -street)
  (or
    (inactive_norm2 ?p ?street1 ?street2)
    (not (and (hasPizzaFor ?p ?street1 ?street2)
              (> (p_time) (goalpizzatime ?street1 ?street2))))))
```

```
122 (:derived (active_norm2-rep ?p -person ?street1 -street ?street2 -street)
      (and
124     (or (and (viol_norm2 ?p ?street1 ?street2)
                 (not (prev_viol_norm2 ?p ?street1 ?street2)))
126         (prev_active_norm2-rep ?p ?street1 ?street2))
        (not (points_augmented ?p))))
128
    (:derived (inactive_norm2-rep ?p -person ?street1 -street ?street2 -street)
130   (or (and (or (or (not (and
                               (or (hasPizzaFor ?p ?street1 ?street2)
132                                    (prev_active_norm2 ?p ?street1 ?street2))
                             (not (pizza_delivered ?p ?street1 ?street2))))
134                        (not (and (hasPizzaFor ?p ?street1 ?street2)
                                   (> (p_time) (goalpizzatime ?street1 ?street2)))))
136                (prev_viol_norm2 ?p ?street1 ?street2))
               (not (prev_active_norm2-rep ?p ?street1 ?street2)))
138       (points_augmented ?p)))

140 ;;;;;;;;;;;;;;;;;; NORM 3;;;;;;;;;;;;;;;;;;;;;;;

142 (:derived (active_norm3 ?p -person ?v -vehicle)
      (driving ?p ?v))
144
    (:derived (inactive_norm3 ?p -person ?v -vehicle)
146   (not (driving ?p ?v)))

148 (:derived (viol_norm3 ?p -person ?v -vehicle)
      (and (active_norm3 ?p ?v)
150        (exists (?street1 -street ?street2 -street)
             (and (vehicle_at ?v ?street1 ?street2)
152               (< (speedLimit ?street1) (vehicleSpeed ?v))))))

154 (:derived (complied-with_norm3 ?p -person ?v -vehicle)
      (or
156     (inactive_norm1 ?p ?v)
        (exists (?street1 ?street2 -street)
158       (and (vehicle_at ?v ?street1 ?street2)
               (> (speedLimit ?street1) (vehicleSpeed ?v))))))
160
    (:derived (active_norm3-rep ?p -person ?v -vehicle)
162   (and (or (and (viol_norm3 ?p ?v) (not (prev_viol_norm3 ?p ?v)))
               (prev_active_norm3-rep ?p ?v))
164        (not (fine_paid3 ?p))))

166 (:derived (inactive_norm3-rep ?p -person ?v -vehicle)
      (or (and (or (or (not (driving ?p ?v))
168                    (exists (?street1 ?street2 -street)
                         (and (vehicle_at ?v ?street1 ?street2)
170                           (> (speedLimit ?street1) (vehicleSpeed ?v)))))
                   (prev_viol_norm3 ?p ?v))
172            (not (prev_active_norm3-rep ?p ?v)))
           (fine_paid3 ?p)))
174
    ;;;;;;;;;;;;;;;;;;;;;;;; ACTIONS ;;;;;;;;;;;;;;;;;;;;;;;
176
    (:action MoveBetweenJunctions
178  :parameters (?v -vehicle ?main -street
                  ?sideStreet1 -street ?sideStreet2 -street ?speed -speed_type)
180  :precondition
      (and
182    (speed_type ?speed)
       (or (vehicle_at ?v ?main ?sideStreet1) (vehicle_at ?v ?sideStreet1 ?main))
```

```
  (or (connection ?main ?sideStreet1 ?sideStreet2)
      (connection ?main ?sideStreet2 ?sideStreet1))
 )
:effect
 (and
  (not (vehicle_at ?v ?main ?sideStreet1))
  (not (vehicle_at ?v ?sideStreet1 ?main))
  (vehicle_at ?v ?main ?sideStreet2)
  (forall (?someStreet -street)
    (when (movedBetweenJunctions ?v ?main ?someStreet ?sideStreet1)
      (not (movedBetweenJunctions ?v ?main ?someStreet ?sideStreet1))))
  (forall (?someStreet -street)
    (when (movedBetweenJunctions ?v ?sideStreet1 ?someStreet ?main)
      (not (movedBetweenJunctions ?v ?sideStreet1 ?someStreet ?main))))
  (movedBetweenJunctions ?v ?main ?sideStreet1 ?sideStreet2)

  ;;(when (= ?speed low) (and (increase (p_time) 2) (assign (vehicleSpeed ?v) 20)))
  (when (= ?speed medium) (and (increase (p_time) 1.5) (assign (vehicleSpeed ?v) 40)))
  (when (= ?speed high) (and (increase (p_time) 1) (assign (vehicleSpeed ?v) 60)))

  (forall (?p -person ?v -vehicle)
    (when (active_norm1 ?p ?v) (prev_active_norm1 ?p ?v)))
  (forall (?p -person ?v -vehicle)
    (when (inactive_norm1 ?p ?v) (not (prev_active_norm1 ?p ?v))))
  (forall (?p -person ?v -vehicle)
    (when (viol_norm1 ?p ?v) (prev_viol_norm1 ?p ?v)))
  (forall (?p -person ?v -vehicle)
    (when (complied-with_norm1 ?p ?v) (not (prev_viol_norm1 ?p ?v))))
  (forall (?p -person ?v -vehicle)
    (when (active_norm1-rep ?p ?v) (prev_active_norm1-rep ?p ?v)))
  (forall (?p -person ?v -vehicle)
    (when (inactive_norm1-rep ?p ?v) (not (prev_active_norm1-rep ?p ?v))))

  (forall (?p -person ?street1 -street ?street2 -street)
    (when (active_norm2 ?p ?street1 ?street2)
          (prev_active_norm2 ?p ?street1 ?street2)))
  (forall (?p -person ?street1 -street ?street2 -street)
    (when (inactive_norm2 ?p ?street1 ?street2)
          (not (prev_active_norm2 ?p ?street1 ?street2))))
  (forall (?p -person ?street1 -street ?street2 -street)
    (when (viol_norm2 ?p ?street1 ?street2)
          (prev_viol_norm2 ?p ?street1 ?street2)))
  (forall (?p -person ?street1 -street ?street2 -street)
    (when (complied-with_norm2 ?p ?street1 ?street2)
          (not (prev_viol_norm2 ?p ?street1 ?street2))))
  (forall (?p -person ?street1 -street ?street2 -street)
    (when (active_norm2-rep ?p ?street1 ?street2)
          (prev_active_norm2-rep ?p ?street1 ?street2)))
  (forall (?p -person ?street1 -street ?street2 -street)
    (when (inactive_norm2-rep ?p ?street1 ?street2)
          (not (prev_active_norm2-rep ?p ?street1 ?street2))))

  (forall (?p -person ?v -vehicle)
    (when (active_norm3 ?p ?v) (prev_active_norm3 ?p ?v)))
  (forall (?p -person ?v -vehicle)
    (when (inactive_norm3 ?p ?v) (not (prev_active_norm3 ?p ?v))))
  (forall (?p -person ?v -vehicle)
    (when (viol_norm3 ?p ?v) (prev_viol_norm3 ?p ?v)))
  (forall (?p -person ?v -vehicle)
    (when (complied-with_norm3 ?p ?v) (not (prev_viol_norm3 ?p ?v))))
  (forall (?p -person ?v -vehicle)
    (when (active_norm3-rep ?p ?v) (prev_active_norm3-rep ?p ?v)))
```

```
    (forall (?p -person ?v -vehicle)
      (when (inactive_norm3-rep ?p ?v) (not (prev_active_norm3-rep ?p ?v))))

    )
)

(:action DeliverPizza
 :parameters (?p -person ?street1 -street ?street2 -street)
 :precondition
   (and
    (exists (?v) (and
     (driving ?p ?v)
     (or (vehicle_at ?v ?street1 ?street2) (vehicle_at ?v ?street2 ?street1))))
   (hasPizzaFor ?p ?street1 ?street2)
 )
 :effect
   (and
     (pizza_delivered ?p ?street1 ?street2)
     (pizza_delivered ?p ?street2 ?street1)
     (not (hasPizzaFor ?p ?street1 ?street2))
     (not (hasPizzaFor ?p ?street2 ?street1))
     (increase (p_time) 3)

    (forall (?p -person ?v -vehicle)
      (when (active_norm1 ?p ?v) (prev_active_norm1 ?p ?v)))
    (forall (?p -person ?v -vehicle)
      (when (inactive_norm1 ?p ?v) (not (prev_active_norm1 ?p ?v))))
    (forall (?p -person ?v -vehicle)
      (when (viol_norm1 ?p ?v) (prev_viol_norm1 ?p ?v)))
    (forall (?p -person ?v -vehicle)
      (when (complied-with_norm1 ?p ?v) (not (prev_viol_norm1 ?p ?v))))
    (forall (?p -person ?v -vehicle)
      (when (active_norm1-rep ?p ?v) (prev_active_norm1-rep ?p ?v)))
    (forall (?p -person ?v -vehicle)
      (when (inactive_norm1-rep ?p ?v) (not (prev_active_norm1-rep ?p ?v))))

    (forall (?p -person ?street1 -street ?street2 -street)
      (when (active_norm2 ?p ?street1 ?street2)
            (prev_active_norm2 ?p ?street1 ?street2)))
    (forall (?p -person ?street1 -street ?street2 -street)
      (when (inactive_norm2 ?p ?street1 ?street2)
            (not (prev_active_norm2 ?p ?street1 ?street2))))
    (forall (?p -person ?street1 -street ?street2 -street)
      (when (viol_norm2 ?p ?street1 ?street2)
            (prev_viol_norm2 ?p ?street1 ?street2)))
    (forall (?p -person ?street1 -street ?street2 -street)
      (when (complied-with_norm2 ?p ?street1 ?street2)
            (not (prev_viol_norm2 ?p ?street1 ?street2))))
    (forall (?p -person ?street1 -street ?street2 -street)
      (when (active_norm2-rep ?p ?street1 ?street2)
            (prev_active_norm2-rep ?p ?street1 ?street2)))
    (forall (?p -person ?street1 -street ?street2 -street)
      (when (inactive_norm2-rep ?p ?street1 ?street2)
            (not (prev_active_norm2-rep ?p ?street1 ?street2))))

    (forall (?p -person ?v -vehicle)
      (when (active_norm3 ?p ?v) (prev_active_norm3 ?p ?v)))
    (forall (?p -person ?v -vehicle)
      (when (inactive_norm3 ?p ?v) (not (prev_active_norm3 ?p ?v))))
    (forall (?p -person ?v -vehicle)
      (when (viol_norm3 ?p ?v) (prev_viol_norm3 ?p ?v)))
    (forall (?p -person ?v -vehicle)
```

```
308       (when (complied-with_norm3 ?p ?v) (not (prev_viol_norm3 ?p ?v))))
        (forall (?p -person ?v -vehicle)
310       (when (active_norm3-rep ?p ?v) (prev_active_norm3-rep ?p ?v)))
        (forall (?p -person ?v -vehicle)
312       (when (inactive_norm3-rep ?p ?v) (not (prev_active_norm3-rep ?p ?v))))
      )
314 )

316 (:action PayFine1
     :parameters (?p -person)
318  :precondition
      (and
320     (exists (?v -vehicle) (driving ?p ?v))
      )
322  :effect
      (and
324     (fine_paid1 ?p)
        (increase (fine ?p) 30)
326
        (forall (?p -person ?v -vehicle)
328       (when (active_norm1 ?p ?v) (prev_active_norm1 ?p ?v)))
        (forall (?p -person ?v -vehicle)
330       (when (inactive_norm1 ?p ?v) (not (prev_active_norm1 ?p ?v))))
        (forall (?p -person ?v -vehicle)
332       (when (viol_norm1 ?p ?v) (prev_viol_norm1 ?p ?v)))
        (forall (?p -person ?v -vehicle)
334       (when (complied-with_norm1 ?p ?v) (not (prev_viol_norm1 ?p ?v))))
        (forall (?p -person ?v -vehicle)
336       (when (active_norm1-rep ?p ?v) (prev_active_norm1-rep ?p ?v)))
        (forall (?p -person ?v -vehicle)
338       (when (inactive_norm1-rep ?p ?v) (not (prev_active_norm1-rep ?p ?v))))

340     (forall (?p -person ?street1 -street ?street2 -street)
          (when (active_norm2 ?p ?street1 ?street2)
342             (prev_active_norm2 ?p ?street1 ?street2)))
        (forall (?p -person ?street1 -street ?street2 -street)
344       (when (inactive_norm2 ?p ?street1 ?street2)
                (not (prev_active_norm2 ?p ?street1 ?street2))))
346     (forall (?p -person ?street1 -street ?street2 -street)
          (when (viol_norm2 ?p ?street1 ?street2)
348             (prev_viol_norm2 ?p ?street1 ?street2)))
        (forall (?p -person ?street1 -street ?street2 -street)
350       (when (complied-with_norm2 ?p ?street1 ?street2)
                (not (prev_viol_norm2 ?p ?street1 ?street2))))
352     (forall (?p -person ?street1 -street ?street2 -street)
          (when (active_norm2-rep ?p ?street1 ?street2)
354             (prev_active_norm2-rep ?p ?street1 ?street2)))
        (forall (?p -person ?street1 -street ?street2 -street)
356       (when (inactive_norm2-rep ?p ?street1 ?street2)
                (not (prev_active_norm2-rep ?p ?street1 ?street2))))
358
        (forall (?p -person ?v -vehicle)
360       (when (active_norm3 ?p ?v) (prev_active_norm3 ?p ?v)))
        (forall (?p -person ?v -vehicle)
362       (when (inactive_norm3 ?p ?v) (not (prev_active_norm3 ?p ?v))))
        (forall (?p -person ?v -vehicle)
364       (when (viol_norm3 ?p ?v) (prev_viol_norm3 ?p ?v)))
        (forall (?p -person ?v -vehicle)
366       (when (complied-with_norm3 ?p ?v) (not (prev_viol_norm3 ?p ?v))))
        (forall (?p -person ?v -vehicle)
368       (when (active_norm3-rep ?p ?v) (prev_active_norm3-rep ?p ?v)))
        (forall (?p -person ?v -vehicle)
```

```
      (when (inactive_norm3-rep ?p ?v) (not (prev_active_norm3-rep ?p ?v))))
  )
)

(:action AugmentPoints
 :parameters (?p -person)
 :effect
  (and
    (points_augmented ?p)
    (increase (penalty_points ?p) 10)

   (forall (?p -person ?v -vehicle)
     (when (active_norm1 ?p ?v) (prev_active_norm1 ?p ?v)))
   (forall (?p -person ?v -vehicle)
     (when (inactive_norm1 ?p ?v) (not (prev_active_norm1 ?p ?v))))
   (forall (?p -person ?v -vehicle)
     (when (viol_norm1 ?p ?v) (prev_viol_norm1 ?p ?v)))
   (forall (?p -person ?v -vehicle)
     (when (complied-with_norm1 ?p ?v) (not (prev_viol_norm1 ?p ?v))))
   (forall (?p -person ?v -vehicle)
     (when (active_norm1-rep ?p ?v) (prev_active_norm1-rep ?p ?v)))
   (forall (?p -person ?v -vehicle)
     (when (inactive_norm1-rep ?p ?v) (not (prev_active_norm1-rep ?p ?v))))

   (forall (?p -person ?street1 -street ?street2 -street)
     (when (active_norm2 ?p ?street1 ?street2)
           (prev_active_norm2 ?p ?street1 ?street2)))
   (forall (?p -person ?street1 -street ?street2 -street)
     (when (inactive_norm2 ?p ?street1 ?street2)
           (not (prev_active_norm2 ?p ?street1 ?street2))))
   (forall (?p -person ?street1 -street ?street2 -street)
     (when (viol_norm2 ?p ?street1 ?street2)
           (prev_viol_norm2 ?p ?street1 ?street2)))
   (forall (?p -person ?street1 -street ?street2 -street)
     (when (complied-with_norm2 ?p ?street1 ?street2)
           (not (prev_viol_norm2 ?p ?street1 ?street2))))
   (forall (?p -person ?street1 -street ?street2 -street)
     (when (active_norm2-rep ?p ?street1 ?street2)
           (prev_active_norm2-rep ?p ?street1 ?street2)))
   (forall (?p -person ?street1 -street ?street2 -street)
     (when (inactive_norm2-rep ?p ?street1 ?street2)
           (not (prev_active_norm2-rep ?p ?street1 ?street2))))

   (forall (?p -person ?v -vehicle)
     (when (active_norm3 ?p ?v) (prev_active_norm3 ?p ?v)))
   (forall (?p -person ?v -vehicle)
     (when (inactive_norm3 ?p ?v) (not (prev_active_norm3 ?p ?v))))
   (forall (?p -person ?v -vehicle)
     (when (viol_norm3 ?p ?v) (prev_viol_norm3 ?p ?v)))
   (forall (?p -person ?v -vehicle)
     (when (complied-with_norm3 ?p ?v) (not (prev_viol_norm3 ?p ?v))))
   (forall (?p -person ?v -vehicle)
     (when (active_norm3-rep ?p ?v) (prev_active_norm3-rep ?p ?v)))
   (forall (?p -person ?v -vehicle)
     (when (inactive_norm3-rep ?p ?v) (not (prev_active_norm3-rep ?p ?v))))

    )
)

(:action PayFine3
 :parameters (?p -person)
 :precondition
```

```
432    (and (exists (?v -vehicle) (driving ?p ?v)))
      :effect
434     (and
        (fine_paid3 ?p)
436     (increase (fine ?p) 30)

438     (forall (?p -person ?v -vehicle)
          (when (active_norm1 ?p ?v) (prev_active_norm1 ?p ?v)))
440     (forall (?p -person ?v -vehicle)
          (when (inactive_norm1 ?p ?v) (not (prev_active_norm1 ?p ?v))))
442     (forall (?p -person ?v -vehicle)
          (when (viol_norm1 ?p ?v) (prev_viol_norm1 ?p ?v)))
444     (forall (?p -person ?v -vehicle)
          (when (complied-with_norm1 ?p ?v) (not (prev_viol_norm1 ?p ?v))))
446     (forall (?p -person ?v -vehicle)
          (when (active_norm1-rep ?p ?v) (prev_active_norm1-rep ?p ?v)))
448     (forall (?p -person ?v -vehicle)
          (when (inactive_norm1-rep ?p ?v) (not (prev_active_norm1-rep ?p ?v))))
450
        (forall (?p -person ?street1 -street ?street2 -street)
452       (when (active_norm2 ?p ?street1 ?street2)
                (prev_active_norm2 ?p ?street1 ?street2)))
454     (forall (?p -person ?street1 -street ?street2 -street)
          (when (inactive_norm2 ?p ?street1 ?street2)
456             (not (prev_active_norm2 ?p ?street1 ?street2))))
        (forall (?p -person ?street1 -street ?street2 -street)
458       (when (viol_norm2 ?p ?street1 ?street2)
                (prev_viol_norm2 ?p ?street1 ?street2)))
460     (forall (?p -person ?street1 -street ?street2 -street)
          (when (complied-with_norm2 ?p ?street1 ?street2)
462             (not (prev_viol_norm2 ?p ?street1 ?street2))))
        (forall (?p -person ?street1 -street ?street2 -street)
464       (when (active_norm2-rep ?p ?street1 ?street2)
                (prev_active_norm2-rep ?p ?street1 ?street2)))
466     (forall (?p -person ?street1 -street ?street2 -street)
          (when (inactive_norm2-rep ?p ?street1 ?street2)
468             (not (prev_active_norm2-rep ?p ?street1 ?street2))))

470     (forall (?p -person ?v -vehicle)
          (when (active_norm3 ?p ?v) (prev_active_norm3 ?p ?v)))
472     (forall (?p -person ?v -vehicle)
          (when (inactive_norm3 ?p ?v) (not (prev_active_norm3 ?p ?v))))
474     (forall (?p -person ?v -vehicle)
          (when (viol_norm3 ?p ?v) (prev_viol_norm3 ?p ?v)))
476     (forall (?p -person ?v -vehicle)
          (when (complied-with_norm3 ?p ?v) (not (prev_viol_norm3 ?p ?v))))
478     (forall (?p -person ?v -vehicle)
          (when (active_norm3-rep ?p ?v) (prev_active_norm3-rep ?p ?v)))
480     (forall (?p -person ?v -vehicle)
          (when (inactive_norm3-rep ?p ?v) (not (prev_active_norm3-rep ?p ?v))))
482     )
    )
484
    (:action GetOffBike
486   :parameters (?p -person ?v -vehicle)
      :precondition
488    (and (driving ?p ?v) )
      :effect
490    (and
         (not (driving ?p ?v))
492      (increase (p_time) 0.5)
             (increase (factor1) 1)
```

```
494
      (forall (?p -person ?v -vehicle)
496     (when (active_norm1 ?p ?v) (prev_active_norm1 ?p ?v)))
      (forall (?p -person ?v -vehicle)
498     (when (inactive_norm1 ?p ?v) (not (prev_active_norm1 ?p ?v))))
      (forall (?p -person ?v -vehicle)
500     (when (viol_norm1 ?p ?v) (prev_viol_norm1 ?p ?v)))
      (forall (?p -person ?v -vehicle)
502     (when (complied-with_norm1 ?p ?v) (not (prev_viol_norm1 ?p ?v))))
      (forall (?p -person ?v -vehicle)
504     (when (active_norm1-rep ?p ?v) (prev_active_norm1-rep ?p ?v)))
      (forall (?p -person ?v -vehicle)
506     (when (inactive_norm1-rep ?p ?v) (not (prev_active_norm1-rep ?p ?v))))

508   (forall (?p -person ?street1 -street ?street2 -street)
        (when (active_norm2 ?p ?street1 ?street2)
510           (prev_active_norm2 ?p ?street1 ?street2)))
      (forall (?p -person ?street1 -street ?street2 -street)
512     (when (inactive_norm2 ?p ?street1 ?street2)
              (not (prev_active_norm2 ?p ?street1 ?street2))))
514   (forall (?p -person ?street1 -street ?street2 -street)
        (when (viol_norm2 ?p ?street1 ?street2)
516           (prev_viol_norm2 ?p ?street1 ?street2)))
      (forall (?p -person ?street1 -street ?street2 -street)
518     (when (complied-with_norm2 ?p ?street1 ?street2)
              (not (prev_viol_norm2 ?p ?street1 ?street2))))
520   (forall (?p -person ?street1 -street ?street2 -street)
        (when (active_norm2-rep ?p ?street1 ?street2)
522           (prev_active_norm2-rep ?p ?street1 ?street2)))
      (forall (?p -person ?street1 -street ?street2 -street)
524     (when (inactive_norm2-rep ?p ?street1 ?street2)
              (not (prev_active_norm2-rep ?p ?street1 ?street2))))
526
      (forall (?p -person ?v -vehicle)
528     (when (active_norm3 ?p ?v) (prev_active_norm3 ?p ?v)))
      (forall (?p -person ?v -vehicle)
530     (when (inactive_norm3 ?p ?v) (not (prev_active_norm3 ?p ?v))))
      (forall (?p -person ?v -vehicle)
532     (when (viol_norm3 ?p ?v) (prev_viol_norm3 ?p ?v)))
      (forall (?p -person ?v -vehicle)
534     (when (complied-with_norm3 ?p ?v) (not (prev_viol_norm3 ?p ?v))))
      (forall (?p -person ?v -vehicle)
536     (when (active_norm3-rep ?p ?v) (prev_active_norm3-rep ?p ?v)))
      (forall (?p -person ?v -vehicle)
538     (when (inactive_norm3-rep ?p ?v) (not (prev_active_norm3-rep ?p ?v))))
    )
540 )
    )
```

FIGURE B.9: Pizza delivery example domain in PDDL

```
(define (problem myproblem)
(:domain pizza_delivery)
(:objects
        paris corcega rosello provenca
        urgell villaroel casanova muntaner - street
        bmw - vehicle
        sergio - person
        medium  high - speed_type
        )

        (:init

        ;;      (speed_type low)
                (speed_type medium)
                (speed_type high)

                (connection muntaner paris corcega)
                (connection muntaner corcega rosello)
                (connection muntaner rosello provenca)

                (connection casanova corcega paris)
                (connection casanova rosello corcega)
                (connection casanova provenca rosello)

                (connection villaroel paris corcega)
                (connection villaroel corcega rosello)
                (connection villaroel rosello provenca)

                (connection urgell corcega paris)
                (connection urgell rosello corcega)
                (connection urgell provenca rosello)

                (connection paris urgell villaroel)
                (connection paris villaroel casanova)
                (connection paris casanova muntaner)

                (connection corcega muntaner casanova)
                (connection corcega casanova villaroel)
                (connection corcega villaroel urgell)

                (connection rosello urgell villaroel)
                (connection rosello casanova muntaner)

                (connection provenca muntaner casanova)
                (connection provenca casanova villaroel)
                (connection provenca villaroel urgell)

                (= (speedLimit paris) 50)
                (= (speedLimit corcega) 50)
                (= (speedLimit rosello) 50)
                (= (speedLimit provenca) 50)
                (= (speedLimit urgell) 50)
                (= (speedLimit villaroel) 50)
                (= (speedLimit casanova) 50)
                (= (speedLimit muntaner) 50)

                (driving sergio bmw)
                (vehicle_at bmw muntaner provenca)
                (= (vehicleSpeed bmw) 10)

                (hasPizzaFor sergio corcega casanova)
                (= (goalpizzatime corcega casanova) 28)
```

```
63
                (hasPizzaFor sergio rosello urgell)
65              (= (goalpizzatime rosello urgell) 19)

67              (= (p_time) 0)
                (= (fine sergio) 0)
69              (= (penalty_points sergio) 0)

71              (movedBetweenJunctions bmw muntaner rosello provenca)

73              (= (factor1) 0)

75              (true)
        )
77
(:goal
79      (and
                (pizza_delivered sergio corcega casanova)
81              (pizza_delivered sergio rosello urgell)

83              (inactive_norm1 sergio bmw)
                (inactive_norm1-rep sergio bmw)
85
                (inactive_norm2-rep sergio corcega casanova)
87              (inactive_norm2-rep sergio rosello urgell)

89              (inactive_norm3-rep sergio bmw)
        )
91 )

93 (:metric
  minimize (+ (+ (* 1 (+ (factor1) (/ (p_time) 30)))
95                 (* 1 (+ (factor1) (/ (penalty_points sergio) 10))))
                 (* 1 (+ (factor1) (/ (fine sergio) 60)))))
97
)
```

FIGURE B.10: Pizza delivery example problem in PDDL

B.4 Pizza Delivery Example 2APL Code

```
Beliefs:
2
        person(sergio).
4       vehicle(bmw).

6       street(paris).
        street(corcega).
8       street(rosello).
        street(provenca).
10      street(urgell).
        street(villaroel).
12      street(casanova).
        street(muntaner).
14
        connection(muntaner, paris, corcega).
16      connection(muntaner, corcega, rosello).
```

```
        connection(muntaner, rosello, provenca).
18
        connection(casanova, corcega, paris).
20      connection(casanova, rosello, corcega).
        connection(casanova, provenca, rosello).
22
        connection(villaroel, paris, corcega).
24      connection(villaroel, corcega, rosello).
        connection(villaroel, rosello, provenca).
26
        connection(urgell, corcega, paris).
28      connection(urgell, rosello, corcega).
        connection(urgell, provenca, rosello).
30
        connection(paris, urgell, villaroel).
32      connection(paris, villaroel, casanova).
        connection(paris, casanova, muntaner).
34
        connection(corcega, muntaner, casanova).
36      connection(corcega, casanova, villaroel).
        connection(corcega, villaroel, urgell).
38
        connection(rosello, urgell, villaroel).
40      connection(rosello, villaroel, urgell).
        connection(rosello, casanova, muntaner).
42
        connection(provenca, muntaner, casanova).
44      connection(provenca, casanova, villaroel).
        connection(provenca, villaroel, urgell).
```

Figure B.11: Pizza delivery example topology in separate 2APL file, topology.2apl

```
   Include:
2          topology.2apl

4  BeliefUpdates:

6  {
    (vehicle_at(V, Main, Street1) or vehicle_at(V, Street1, Main)) and
8   (connection(Main, Street1, Street2) or connection(Main, Street2, Street1)) and
    p_time(T)
10 }
   MoveBetweenJunctionsLow(V, Main, Street1, Street2, Speed)
12 {
    not vehicle_at(V, Main, Street1),
14  not vehicle_at(V, Street1, Main),
    vehicle_at(V, Main, Street2),
16  not movedBetweenJunctions(V, Main, paris, Street1),
    not movedBetweenJunctions(V, Main, corcega, Street1),
18  not movedBetweenJunctions(V, Main, rosello, Street1),
    not movedBetweenJunctions(V, Main, provenca, Street1),
20  not movedBetweenJunctions(V, Main, urgell, Street1),
    not movedBetweenJunctions(V, Main, villaroel, Street1),
22  not movedBetweenJunctions(V, Main, casanova, Street1),
    not movedBetweenJunctions(V, Main, muntaner, Street1),
24  not movedBetweenJunctions(V, Street1, paris, Main),
    not movedBetweenJunctions(V, Street1, corcega, Main),
26  not movedBetweenJunctions(V, Street1, rosello, Main),
    not movedBetweenJunctions(V, Street1, provenca, Main),
```

```
 not movedBetweenJunctions(V, Street1, urgell, Main),
 not movedBetweenJunctions(V, Street1, villaroel, Main),
 not movedBetweenJunctions(V, Street1, casanova, Main),
 not movedBetweenJunctions(V, Street1, muntaner, Main),
 movedBetweenJunctions(V, Main, Street1, Street2),
 p_time(T+2), not p_time(T)
}

{
 (vehicle_at(V, Main, Street1) or vehicle_at(V, Street1, Main)) and
 (connection(Main, Street1, Street2) or connection(Main, Street2, Street1)) and
 p_time(T)
}
MoveBetweenJunctionsMedium(V, Main, Street1, Street2, Speed)
{
 not vehicle_at(V, Main, Street1),
 not vehicle_at(V, Street1, Main),
 vehicle_at(V, Main, Street2),
 not movedBetweenJunctions(V, Main, paris, Street1),
 not movedBetweenJunctions(V, Main, corcega, Street1),
 not movedBetweenJunctions(V, Main, rosello, Street1),
 not movedBetweenJunctions(V, Main, provenca, Street1),
 not movedBetweenJunctions(V, Main, urgell, Street1),
 not movedBetweenJunctions(V, Main, villaroel, Street1),
 not movedBetweenJunctions(V, Main, casanova, Street1),
 not movedBetweenJunctions(V, Main, muntaner, Street1),
 not movedBetweenJunctions(V, Street1, paris, Main),
 not movedBetweenJunctions(V, Street1, corcega, Main),
 not movedBetweenJunctions(V, Street1, rosello, Main),
 not movedBetweenJunctions(V, Street1, provenca, Main),
 not movedBetweenJunctions(V, Street1, urgell, Main),
 not movedBetweenJunctions(V, Street1, villaroel, Main),
 not movedBetweenJunctions(V, Street1, casanova, Main),
 not movedBetweenJunctions(V, Street1, muntaner, Main),
 movedBetweenJunctions(V, Main, Street1, Street2),
 p_time(T+1.5), not p_time(T)
}

{
 (vehicle_at(V, Main, Street1) or vehicle_at(V, Street1, Main)) and
 (connection(Main, Street1, Street2) or connection(Main, Street2, Street1)) and
 p_time(T)
}
MoveBetweenJunctionsHigh(V, Main, Street1, Street2, Speed)
{
 not vehicle_at(V, Main, Street1),
 not vehicle_at(V, Street1, Main),
 vehicle_at(V, Main, Street2),
 not movedBetweenJunctions(V, Main, paris, Street1),
 not movedBetweenJunctions(V, Main, corcega, Street1),
 not movedBetweenJunctions(V, Main, rosello, Street1),
 not movedBetweenJunctions(V, Main, provenca, Street1),
 not movedBetweenJunctions(V, Main, urgell, Street1),
 not movedBetweenJunctions(V, Main, villaroel, Street1),
 not movedBetweenJunctions(V, Main, casanova, Street1),
 not movedBetweenJunctions(V, Main, muntaner, Street1),
 not movedBetweenJunctions(V, Street1, paris, Main),
 not movedBetweenJunctions(V, Street1, corcega, Main),
 not movedBetweenJunctions(V, Street1, rosello, Main),
 not movedBetweenJunctions(V, Street1, provenca, Main),
 not movedBetweenJunctions(V, Street1, urgell, Main),
 not movedBetweenJunctions(V, Street1, villaroel, Main),
```

```
90  not movedBetweenJunctions(V, Street1, casanova, Main),
    not movedBetweenJunctions(V, Street1, muntaner, Main),
92  movedBetweenJunctions(V, Main, Street1, Street2),
    p_time(T+1), not p_time(T)
94 }

96 {
    driving(P, V) and
98  (vehicle_at(V, Street1, Street2) or (vehicle_at(V, Street2, Street1))) and
    hasPizzaFor(P,Street1,Street2) and
100 p_time(T)
   }
102 DeliverPizza(P,Street1,Street2)
   {
104 pizza_delivered(P,Street1,Street2),
    pizza_delivered(P,Street2,Street1),
106 not hasPizzaFor(P,Street1,Street2),
    not hasPizzaFor(P,Street2,Street1),
108 p_time(T+3), not p_time(T)
   }
110
   {
112 driving(P,V) and
    p_time(T)
114 }
   GetOffBike(P,V)
116 {
    not driving(P,V),
118 p_time(T+0.5), not p_time(T)
   }
120
   {
122 fine(P, M) and
    not fine_paid1(P)
124 }
   PayFine1(P)
126 {
    fine_paid1(P),
128 fine(P, M+30),
    not fine(P, M)
130 }

132 {
    fine(P, M) and
134 not fine_paid3(P)
   }
136 PayFine3(P)
   {
138 fine_paid3(P),
    fine(P, M+30),
140 not fine(P, M)
   }
142
   {
144 penalty_points(P, Pts) and
    not points_augmented(P)
146 }
   AugmentPoints(P)
148 {
    points_augmented(P),
150 penalty_points(P, Pts+10),
    not penalty_points(P, Pts)
```

```
152 }

154 Beliefs:

156 driving(sergio, bmw).
    vehicle_at(bmw, muntaner, provenca).
158 vehicleSpeed(bmw, 10).
    hasPizzaFor(sergio, corcega, casanova).
160 goalpizzatime(corcega, casanova, 28).
    hasPizzaFor(sergio, rosello, urgell).
162 goalpizzatime(rosello, urgell, 19).
    p_time(0).
164 fine(sergio,0).
    penalty_points(sergio,0).
166 movedBetweenJunctions(bmw, muntaner, rosello, provenca).

168 Plans:
    @pizzaworld(enter(sergio, muntaner, provenca))
170
    Goals:
172 pizza_delivered(sergio, corcega, casanova) and
    pizza_delivered(sergio, rosello, urgell)
174
    PC-rules:
176
    execute_plan(movebetweenjunctions(V, Main, Street1, Street2, Speed))<-true|
178 {
     if B(Speed=low) then
180  {
            MoveBetweenJunctionsLow(V, Main, Street1, Street2, Speed);
182         @pizzaworld(moveBetweenJunctions(V, Main, Street1, Street2, low))
     }
184  else if B(Speed=medium) then
     {
186         MoveBetweenJunctionsMedium(V, Main, Street1, Street2, Speed);
            @pizzaworld(moveBetweenJunctions(V, Main, Street1, Street2, medium))
188  }
     else
190  {
    MoveBetweenJunctionsHigh(V, Main, Street1, Street2, Speed);
192 @pizzaworld(moveBetweenJunctions(V, Main, Street1, Street2, high))
     }
194 }

196 execute_plan(deliverpizza(P, Street1, Street2))<-true|
    {
198  DeliverPizza(P,Street1,Street2);
     @pizzaworld(deliverPizza(P,Street1,Street2))
200 }

202 execute_plan(getoffbike(P, V))<-true|
    {
204  GetOffBike(P, V);
     @pizzaworld(getOffBike(P, V))
206 }

208 execute_plan(payfine1(P))<-true|
    {
210  PayFine1(P);
     @pizzaworld(payFine1(P))
212 }
```

```
214 execute_plan(payfine3(P))<-true|
    {
216     PayFine3(P);
     @pizzaworld(payFine3(P))
218 }

220 execute_plan(augmentpoints(P))<-true|
    {
222  AugmentPoints(P);
     @pizzaworld(augmentpoints(P))
224 }
```

FIGURE B.12: Pizza delivery agent in 2APL, main_agent.2apl

Appendix C

Proofs

This appendix provides the proofs for the deontic logic reductions of our normative framework described in Section 4.4. The purpose of these proofs is to demonstrate that our norm representation (a set of first-order logic formulas) can represent both deontic statements in Standard Deontic Logic and in Dyadic Logic.

C.1 Achievement Obligations

In our framework, achievement obligations (typically, $O(A)$ where A is a state to be eventually achieved once in the future) are characterised by leaving the discharge condition as the unique free parameter (see Section 4.4.1):

$$\langle \pi, i, \theta \rangle \models \mathcal{O}_{\theta f_n^R \leq timeout_n}(E_\alpha[\theta\top] \preceq \theta f_n^D \mid \theta\top) \text{ iff } \langle \pi, i, \theta \rangle \models \exists j \geq i, \theta'' : \langle \pi, j, \theta\theta'' \rangle \models f_n^D$$

Proof of this equality is given in Section C.1.1. In the case of achievement obligations, axioms K (subsection C.1.2) and *Necessitation* (subsection C.1.3) can be proven.

A relevant issue of achievement obligations in our framework is that axiom D is not fulfilled. The reason is that the negation of the reduction is simply $\neg f_n^D$, that is, that the state to be achieved is the complementary state of the original achievement obligation. The main implication of this is that, in our framework, when dealing with achievement obligations as commonly treated in SDL, we cannot ensure that $O(A)$ and $O(\neg A)$ are incompatible. While from a theoretical perspective this might seem a problem, from a practical point of view trying to ensure this property above others might be an even bigger problem: if the only thing we care about is to achieve two goal states and we do not care about maintenance, then it is not really a drawback if both obligations can be achieved at different points of time.

For simplification purposes in the subsequent proofs, we will assume that:

$$\mathcal{O}(f_n^D) \equiv \mathcal{O}_{\theta f_n^R \leq timeout_n}(E_\alpha[\theta\top] \preceq \theta f_n^D \mid \theta\top)$$

C.1.1 Substitution for Achievement Obligations

$$
\begin{aligned}
&\langle \pi,i,\theta \rangle \models \mathcal{O}_{\theta f_n^R \leq timeout_n}(E_\alpha[\theta f_n^M] \preceq \theta f_n^D \mid \theta f_n^A)\\
\text{iff } &\langle \pi,i,\theta \rangle \models \mathbf{G}(\neg f_n^A) \vee\\
&\quad \left[\neg f_n^A \mathbf{U}\left(f_n^A \wedge [\forall\theta' : \theta' f_n^M \mathbf{U} \exists\theta'' : \theta'' f_n^D]\right)\right] \vee\\
&\quad \left[\neg f_n^A \mathbf{U}\left(f_n^A \wedge [\exists\theta' : \theta' f_n^M \mathbf{U}(\neg\theta' f_n^M \wedge [\exists\theta'' : \mathbf{F}_{\leq timeout_n} \theta''\theta' f_n^R])]\right)\right]\\
\text{iff } &\langle \pi,i,\theta \rangle \models \mathbf{G}(\neg\top) \vee\\
&\quad \left[\neg\top \mathbf{U}\left(\top \wedge [\forall\theta' : \theta' \top \mathbf{U} \exists\theta'' : \theta'' f_n^D]\right)\right] \vee\\
&\quad \left[\neg\top \mathbf{U}\left(\top \wedge [\exists\theta' : \theta' \top \mathbf{U}(\neg\theta' \top \wedge [\exists\theta'' : \mathbf{F}_{\leq timeout_n} \theta''\theta' \bot]\right.\right.\\
\text{iff } &\langle \pi,i,\theta \rangle \models \mathbf{G}(\bot) \vee\\
&\quad \left[\mathbf{F}(\exists\theta'' : \theta'' f_n^D)\right] \vee\\
&\quad \left[\exists\theta' : \theta' \top \mathbf{U}(\bot \wedge [\bot])\right]\\
\text{iff } &\langle \pi,i,\theta \rangle \models \left[\mathbf{F}(\exists\theta'' : \theta'' f_n^D)\right]\\
\text{iff}_{Def\ 4.1.(g)}\ &\exists j \geq i,\theta'' : \langle \pi,i,\theta'\theta \rangle \models f_n^D
\end{aligned}
$$

The intuition behind this result is that in an achievement obligation, the only thing that matters is that the goal state is eventually achieved. In a norm of this kind the activating condition plays no role, as the obligation stands from the moment it is announced. There is also no maintenance to be done, as achieving the goal state does not depend on the previous states to the actual achievement.

C.1.2 Proof of K

Axiom K states that $\mathcal{O}(A \rightarrow B) \rightarrow (\mathcal{O}(A) \rightarrow \mathcal{O}(B))$.

$$
\begin{aligned}
&\langle \pi,i,\theta \rangle \models \mathcal{O}(A \rightarrow B)\\
\text{iff } &\exists j \geq i,\theta' : \langle \pi,j,\theta'\theta \rangle \models (A \rightarrow B)\\
\text{iff } &\exists j \geq i,\theta' : \langle \pi,j,\theta'\theta \rangle \models (\neg A \vee B)\\
\text{iff } &\exists j \geq i,\theta' : \langle \pi,j,\theta'\theta \rangle \models \neg A \vee B\\
\text{iff } &\exists j \geq i,\theta' : (\langle \pi,j,\theta'\theta \rangle \models \neg A \vee \langle \pi,j,\theta'\theta'' \rangle \models B)\\
\text{iff } &\exists j \geq i,\theta' : (\langle \pi,j,\theta'\theta \rangle \models \neg A) \vee \langle \pi,j,\theta'\theta'' \rangle \models B)\\
\text{iff } &\exists j \geq i,\theta' : (\langle \pi,j,\theta'\theta \rangle \not\models A \vee \langle \pi,j,\theta'\theta'' \rangle \models B)\\
\text{iff } &(\exists j \geq i,\theta' : \langle \pi,j,\theta'\theta' \rangle \not\models A) \vee (\exists j \geq i,\theta' : \langle \pi,j,\theta'\theta \rangle \models B)\\
\text{iff } &(\exists j \geq i,\theta' : \langle \pi,j,\theta'\theta \rangle \not\models A) \vee (\exists j \geq i,\theta' : \langle \pi,j,\theta'\theta \rangle \models B)\\
\text{iff } &\langle \pi,i,\theta \rangle \models \neg\mathcal{O}(A) \vee \mathcal{O}(B)\\
\text{iff } &\langle \pi,i,\theta \rangle \models \mathcal{O}(A) \rightarrow \mathcal{O}(B)
\end{aligned}
$$

Thus, in our framework, $\mathcal{O}(A \rightarrow B) \equiv \mathcal{O}(A) \rightarrow \mathcal{O}(B)$, so K is fulfilled.

C.1.3 Proof of Necessitation

Necessitation states that $\models \alpha \rightarrow \langle \pi,i,\theta \rangle \models \mathcal{O}(\alpha)$: if something is a tautology, then it is obliged for it to happen.

$$
\begin{aligned}
&\models \alpha \\
&\Rightarrow \forall \pi', i', \theta' : \langle \pi', i', \theta' \rangle \models \alpha \\
&\Rightarrow_{\pi'=\pi} \forall i', \theta' : \langle \pi, i', \theta' \rangle \models \alpha \\
&\Rightarrow \forall i', \exists \theta' : \langle \pi, i', \theta' \rangle \models \alpha \\
&\Rightarrow \exists j \geq i, \theta' : \langle \pi, j, \theta' \rangle \models \alpha \\
&\text{iff } \langle \pi, i, \theta \rangle \models \mathcal{O}(\alpha)
\end{aligned}
$$

C.2 Maintenance Obligations

Maintenance obligations are seen in the literature in a very similar form to achievement obligations: $O(A)$, but in this case A is a state to be permanently maintained. In order to represent this in our framework, the maintenance condition has to be the unique free parameter (see Section 4.4.2):

$$\langle \pi, i, \theta \rangle \models \mathcal{O}_{\theta f_n^R \leq timeout_n}(E_\alpha[\theta f_n^M] \preceq \theta f_n^D \mid \theta f_n^A) \text{ iff } \langle \pi, i, \theta \rangle \models \forall \theta' : \mathbf{G}(\theta' f_n^M)$$

Proof of this equality is given in Section C.2.1. Interestingly, axioms K (subsection C.2.2), D (subsection C.2.3) and *Necessitation* (subsection C.2.4) can be proven. Therefore, maintenance obligations in our framework are equivalent to maintenance obligations in SDL.

For simplification purposes in the subsequent proofs, we will assume that:

$$\mathcal{O}(f_n^M) \equiv \mathcal{O}_{\theta f_n^R \leq timeout_n}(E_\alpha[\theta f_n^M] \preceq \theta f_n^D \mid \theta f_n^A)$$

C.2.1 Substitution for Maintenance Obligations

$$
\begin{aligned}
&\quad \langle \pi, i, \theta \rangle \models \mathcal{O}_{\theta f_n^R \leq timeout_n}(E_\alpha[\theta f_n^M] \preceq \theta f_n^D \mid \theta f_n^A) \\
&\text{iff } \langle \pi, i, \theta \rangle \models \mathbf{G}(\neg f_n^A) \vee \\
&\qquad \Big[\neg f_n^A \mathbf{U} \Big(f_n^A \wedge [\forall \theta' : \theta' f_n^M \mathbf{U} \exists \theta'' : \theta'' f_n^D] \Big)\Big] \vee \\
&\qquad \Big[\neg f_n^A \mathbf{U} \Big(f_n^A \wedge [\exists \theta' : \theta' f_n^M \mathbf{U} (\neg \theta' f_n^M \wedge [\exists \theta'' : \mathbf{F}_{\leq timeout_n} \theta'' \theta' f_n^R])] \Big)\Big] \\
&\text{iff } \langle \pi, i, \theta \rangle \models \mathbf{G}(\neg \top) \vee \\
&\qquad \Big[\neg \top \mathbf{U} \Big(\top \wedge [\forall \theta' : \theta' f_n^M \mathbf{U} \exists \theta'' : \theta'' \infty] \Big)\Big] \vee \\
&\qquad \Big[\neg \top \mathbf{U} \Big(\top \wedge [\exists \theta' : \theta' f_n^M \mathbf{U} (\neg \theta' f_n^M \wedge [\exists \theta'' : \mathbf{F}_{\leq timeout_n} \theta'' \theta' \bot])] \Big)\Big] \\
&\text{iff } \langle \pi, i, \theta \rangle \models \mathbf{G}(\bot) \vee \\
&\qquad \Big[[\forall \theta' : \theta' f_n^M \mathbf{U} \exists \theta'' : \theta'' \infty]\Big] \vee \\
&\qquad \Big[[\exists \theta' : \theta' f_n^M \mathbf{U} (\neg \theta' f_n^M \wedge \bot)]\Big] \\
&\text{iff } \langle \pi, i, \theta \rangle \models \Big[\forall \theta' : \theta' f_n^M \mathbf{U} \exists \theta'' : \theta'' \infty\Big] \\
&\text{iff } \langle \pi, i, \theta \rangle \models \Big[\forall \theta' : \mathbf{G}(\theta' f_n^M)\Big] \\
&\text{iff } \forall j \geq i, \theta' : \langle \pi, j, \theta' \theta \rangle \models f_n^M
\end{aligned}
$$

The intuition behind this result is that in a maintenance obligation, the only thing that matters is that the state is maintained until the end of time. In a norm of this kind

the activating condition plays no role, as the obligation stands from the moment it is announced. There is also no discharge to be considered.

C.2.2 Proof of K

Axiom K states that $\mathcal{O}(A \to B) \to (\mathcal{O}(A) \to \mathcal{O}(B))$.

$\langle \pi,i,\theta \rangle \models [\mathcal{O}(A \to B) \wedge \mathcal{O}(A)]$
iff $\langle \pi,i,\theta \rangle \models [\forall \theta' : \mathbf{G}(\theta'(A \to B))] \wedge [\forall \theta' : \mathbf{G}(\theta' A)]$
iff $\langle \pi,i,\theta \rangle \models [\forall \theta' : \mathbf{G}(\theta'(A \to B)) \wedge \mathbf{G}(\theta' A)]$
iff $\langle \pi,i,\theta \rangle \models [\forall \theta' : \mathbf{G}(\theta'(A \to B \wedge A)]$
iff $\langle \pi,i,\theta \rangle \models [\forall \theta' : \mathbf{G}(\theta'(B)]$
iff $\langle \pi,i,\theta \rangle \models \mathcal{O}(B)$

The result is identical to the case of achievement obligations: $\mathcal{O}(A \to B) \equiv \mathcal{O}(A) \to \mathcal{O}(B)$, so K is fulfilled.

C.2.3 Proof of D

Axiom D states that $\mathcal{O}(A) \to \neg\mathcal{O}(\neg A)$.

$\langle \pi,i,\theta \rangle \models \mathcal{O}(A)$
iff $\langle \pi,i,\theta \rangle \models \forall \theta' : \mathbf{G}(\theta' A)$
$\Rightarrow \langle \pi,i,\theta \rangle \models \exists \theta' : \mathbf{G}(\theta' A)$
$\Rightarrow_{\mathbf{G}(A) \to \neg\mathbf{G}(\neg A)} \langle \pi,i,\theta \rangle \models \exists \theta' : \neg\mathbf{G}(\neg\theta' A)$
iff $\langle \pi,i,\theta \rangle \models \neg\forall \theta' : \mathbf{G}(\neg\theta' A)$
iff $\langle \pi,i,\theta \rangle \models \neg\forall \theta' : \mathbf{G}(\theta' \neg A)$
iff $\langle \pi,i,\theta \rangle \models \neg\left[\forall \theta' : \mathbf{G}(\theta' \neg A)\right]$
iff $\langle \pi,i,\theta \rangle \models \neg\mathcal{O}(\neg A)$

In achievement obligations, D cannot be proven, but this is not the case with maintenance obligations. This result is reasonable: because maintenance obligations require that states are fulfilled at all points of time, two obligations with complementary maintenance states are incompatible.

C.2.4 Proof of Necessitation

Necessitation states that $\models \alpha \to \langle \pi,i,\theta \rangle \models \mathcal{O}(\alpha)$: if something is a tautology, then it is obliged for it to happen.

$\models \alpha$
$\Rightarrow \forall \pi',i',\theta' : \langle \pi',i',\theta' \rangle \models \alpha$
$\Rightarrow_{\pi'=\pi} \forall i',\theta' : \langle \pi,i',\theta' \rangle \models \alpha$

$\Rightarrow \forall i', \theta' : \langle \pi, i', \theta' \rangle \models \alpha$
$\Rightarrow \forall j \geq i, \theta' : \langle \pi, j, \theta' \rangle \models \alpha$
iff $\langle \pi, i, \theta \rangle \models \mathcal{O}(\alpha)$

C.3 Dyadic Deontic Logic

Dyadic Deontic Logic (DDL) [Prakken and Sergot, 1997] is an extension of SDL that allows to model conditional obligations, in which the common form is $O(A|\ B)$, read as: "given that B, it is obliged that A", or otherwise, "in the context defined by the event B, it is obliged that A". It is not clear in the literature whether B triggers the context or is required at all times to maintain the context [Prakken and Sergot, 1996], but it is of little importance in our case to define the reduction.

In the first case:

- $f_n^A \equiv B$
- $f_n^M \equiv A$
- $f_n^D \equiv G(f_n^M)$
- $f_n^R \equiv \bot$

And the reduction is (shown in C.3.1):

$$\Big[\forall j \ \geq i : \langle \pi, j, \theta \rangle \models \neg B \vee (\exists j \geq i : \langle \pi, j, \theta \rangle \models B \wedge \forall j \geq i, \theta' : \langle \pi, j, \theta' \theta \rangle \models A\Big] \\ \wedge \forall i \leq k < j, \langle \pi, k, \theta \rangle \models \neg B$$

In the second case:

- $f_n^A \equiv B$
- $f_n^M \equiv (A \wedge B)$
- $f_n^D \equiv \neg B$
- $f_n^R \equiv \bot$

The reduction formula is not explored in this Section but is trivially achieved in an identical fashion:

$$\Big[\forall j \ \geq i : \langle \pi, j, \theta \rangle \models \neg B \vee \\ (\exists j \geq i : \langle \pi, j, \theta \rangle \models B \wedge \exists k : [\forall k > j \geq i, \theta' : \langle \pi, j, \theta' \theta \rangle \models (A \wedge B) \wedge \exists \theta' : \langle \pi, k, \theta' \theta \rangle \models \neg B]\Big] \\ \wedge \forall i \leq k < j, \langle \pi, k, \theta \rangle \models \neg B$$

In this thesis we focus on the first case, but it is trivially provable that the axioms are equally valid on both systems. There exist several of such axiom systems that apply to DDL, but in this thesis we choose the $K1$-$K4$ of [Hilpinen, 1971] and we prove $K1$, $K3$ and $K4$ in subsections C.3.2, C.3.4 and C.3.5.

Axiom $K2$, which is $\neg(\mathcal{O}(A|B) \wedge \mathcal{O}(\neg A|B))$, cannot be reduced to $\top$ in our framework, but to $\mathbf{F}(B)$, as seen in subsection C.3.3. This result, far from discouraging, is actually intuitive. If the condition never happens, can it be stated that it is not valid that $\neg\mathcal{O}(A|B)$ and $\mathcal{O}(A|B)$ coexist? If we take as an example the extreme case: $B \equiv \bot$, is it really a contradiction? For us, it is not: complementary maintenance conditions are only inconsistent if it is a fact that the activating condition is eventually going to hold. For further discussion, please refer to Section 4.7.

C.3.1 Substitution for Dyadic Deontic Logic

$$
\begin{aligned}
&\langle \pi,i,\theta\rangle \models \mathcal{O}_{\theta f_n^R \leq timeout_n}(E_\alpha[\theta f_n^M] \preceq \theta f_n^D \mid \theta f_n^A)\\
\text{iff } &\langle \pi,i,\theta\rangle \models \mathbf{G}(\neg f_n^A) \vee\\
&\qquad \left[\neg f_n^A \mathbf{U}\left(f_n^A \wedge [\forall\theta' : \theta' f_n^M \mathbf{U} \exists\theta'' : \theta'' f_n^D]\right)\right] \vee\\
&\qquad \left[\neg f_n^A \mathbf{U}\left(f_n^A \wedge [\exists\theta' : \theta' f_n^M \mathbf{U}(\neg\theta' f_n^M \wedge [\exists\theta'' : \mathbf{F}_{\leq timeout_n}\theta''\theta' f_n^R])]\right)\right]\\
\text{iff } &\langle \pi,i,\theta\rangle \models \mathbf{G}(\neg f_n^A) \vee\\
&\qquad \left[\neg f_n^A \mathbf{U}\left(f_n^A \wedge [\forall\theta' : \theta' f_n^M \mathbf{U} \exists\theta'' : \theta'' \mathbf{G}(\forall\theta' : \theta' f_n^M)]\right)\right] \vee\\
&\qquad \left[\neg f_n^A \mathbf{U}\left(f_n^A \wedge [\exists\theta' : \theta' f_n^M \mathbf{U}(\neg\theta' f_n^M \wedge [\exists\theta'' : \mathbf{F}_{\leq timeout_n}\theta''\theta' \bot])]\right)\right]\\
\text{iff } &\langle \pi,i,\theta\rangle \models \mathbf{G}(\neg f_n^A) \vee \left[\neg f_n^A \mathbf{U}\left(f_n^A \wedge \mathbf{G}(\forall\theta' : \theta' f_n^M)\right)\right]\\
\text{iff } &\langle \pi,i,\theta\rangle \models (\neg f_n^A \mathbf{U}\mathbf{G}(\neg f_n^A)) \vee \left[\neg f_n^A \mathbf{U}\left(f_n^A \wedge \mathbf{G}(\forall\theta' : \theta' f_n^M)\right)\right]\\
\text{iff } &\langle \pi,i,\theta\rangle \models \neg f_n^A \mathbf{U}\left[\mathbf{G}(\neg f_n^A) \vee \left(f_n^A \wedge \mathbf{G}(\forall\theta' : \theta' f_n^M)\right)\right]\\
\text{iff } &\exists j \geq i : \langle \pi,j,\theta\rangle \models \left[\mathbf{G}(\neg f_n^A) \vee \left(f_n^A \wedge \mathbf{G}(\forall\theta' : \theta' f_n^M)\right)\right] \wedge \forall i \leq k < j, \langle \pi,k,\theta\rangle \models \neg f_n^A\\
\text{iff } &\left[\exists j \geq i : \langle \pi,j,\theta\rangle \models \mathbf{G}(\neg f_n^A) \vee \exists j \geq i : \langle \pi,j,\theta\rangle \models \left(f_n^A \wedge \mathbf{G}(\forall\theta' : \theta' f_n^M)\right)\right] \wedge \forall i \leq k < j, \langle \pi,k,\theta\rangle \models \neg f_n^A\\
\text{iff } &\left[\forall j \ \geq i : \exists j \geq i : \langle \pi,j,\theta\rangle \models \neg f_n^A \vee\right.\\
&\left.(\exists j \geq i : \langle \pi,j,\theta\rangle \models f_n^A \wedge \exists j \geq i : \langle \pi,j,\theta\rangle \models \mathbf{G}(\forall\theta' : \theta' f_n^M)\right] \wedge \forall i \leq k < j, \langle \pi,k,\theta\rangle \models \neg f_n^A\\
\text{iff } &\left[\forall j \ \geq i : \langle \pi,j,\theta\rangle \models \neg f_n^A \vee (\exists j \geq i : \langle \pi,j,\theta\rangle \models f_n^A \wedge \forall j \geq i : \langle \pi,j,\theta\rangle \models (\forall\theta' : \theta' f_n^M)\right]\\
&\wedge \forall i \leq k < j, \langle \pi,k,\theta\rangle \models \neg f_n^A\\
\text{iff } &\left[\forall j \ \geq i : \langle \pi,j,\theta\rangle \models \neg f_n^A \vee (\exists j \geq i : \langle \pi,j,\theta\rangle \models f_n^A \wedge \forall j \geq i, \theta' : \langle \pi,j,\theta'\theta\rangle \models f_n^M)\right]\\
&\wedge \forall i \leq k < j, \langle \pi,k,\theta\rangle \models \neg f_n^A
\end{aligned}
$$

The resulting formula can be informally read as: "either the norm is never activated or else it is activated at some point for the first time and the maintenance condition is always fulfilled from that moment on".

C.3.2 Proof of K1

Axiom $K1$ states that $\mathcal{O}(A \vee \neg A|B)$, that is, that whichever the activating condition is, an obligation which contains both a maintenance condition and its complementary formula is a tautology.

$$
\begin{aligned}
&\langle \pi,i,\theta\rangle \models \mathcal{O}(A \vee \neg A|B)\\
\text{iff } &\left[\forall j \ \geq i : \langle \pi,j,\theta\rangle \models \neg B \vee (\exists j \geq i : \langle \pi,j,\theta\rangle \models B \wedge \forall j \geq i, \theta' : \langle \pi,j,\theta'\theta\rangle \models (A \vee \neg A))\right] \wedge \forall i \leq k < j, \langle \pi,k,\theta\rangle \models \neg B\\
\text{iff } &\left[\forall j \ \geq i : \langle \pi,j,\theta\rangle \models \neg B \vee (\exists j \geq i : \langle \pi,j,\theta\rangle \models B \wedge \forall j \geq i, \theta' : \langle \pi,j,\theta'\theta\rangle \models \top)\right] \wedge \forall i \leq k < j, \langle \pi,k,\theta\rangle \models \neg B
\end{aligned}
$$

$$\text{iff } \Big[\forall j \geq i : \langle \pi, j, \theta \rangle \models \neg B \lor (\exists j \geq i : \langle \pi, j, \theta \rangle \models B \land \top)\Big] \land \forall i \leq k < j, \langle \pi, k, \theta \rangle \models \neg B$$

At this point we are left with two cases: the obligation gets activated or not. In both cases, the formula is a tautology:

If $\mathbf{G}(\neg B)$:

$$\Big[\forall j \geq i : \langle \pi, j, \theta \rangle \models \neg B \lor (\exists j \geq i : \langle \pi, j, \theta \rangle \models B \land \top)\Big] \land \forall i \leq k < j, \langle \pi, k, \theta \rangle \models \neg B$$
$$\text{iff } \Big[\top \lor (\bot \land \top)\Big] \land \top$$
$$\text{iff } \top$$

If $\mathbf{F}(B)$:

$$\Big[\forall j \geq i : \langle \pi, j, \theta \rangle \models \neg B \lor (\exists j \geq i : \langle \pi, j, \theta \rangle \models B \land \top)\Big] \land \forall i \leq k < j, \langle \pi, k, \theta \rangle \models \neg B$$
$$\text{iff } \Big[\forall j \geq i : \langle \pi, j, \theta \rangle \models \neg B \lor (\bot \land \top)\Big] \land \top$$
$$\text{iff } \top$$

C.3.3 Proof of K2

Axiom $K2$ states that $\neg(\mathcal{O}(A|B) \land \mathcal{O}(\neg A|B))$, that is, that it cannot be the case that we have two obligations with the same activating condition and complementary maintenance obligations. This can be seen as the D axiom of SDL with $B \equiv \top$.

$$\langle \pi, i, \theta \rangle \models \neg(\mathcal{O}(A|B) \land \mathcal{O}(\neg A|B))$$
$$\text{iff } \neg\Bigg[\bigg(\Big[\forall j \geq i : \langle \pi, j, \theta \rangle \models \neg B \lor (\exists j \geq i : \langle \pi, j, \theta \rangle \models B \land \forall j \geq i, \theta' : \langle \pi, j, \theta'\theta \rangle \models A)\Big] \land \forall i \leq k < j, \langle \pi, k, \theta \rangle \models \neg B\bigg)$$
$$\land$$
$$\bigg(\Big[\forall j \geq i : \langle \pi, j, \theta \rangle \models \neg B \lor (\exists j \geq i : \langle \pi, j, \theta \rangle \models B \land \forall j \geq i, \theta' : \langle \pi, j, \theta'\theta \rangle \models \neg A)\Big] \land \forall i \leq k < j, \langle \pi, k, \theta \rangle \models \neg B\bigg)\Bigg]$$
$$\text{iff } \neg\bigg(\Big[\forall j \geq i : \langle \pi, j, \theta \rangle \models \neg B \lor (\exists j \geq i : \langle \pi, j, \theta \rangle \models B \land \forall j \geq i, \theta' : \langle \pi, j, \theta'\theta \rangle \models A)\Big] \land \forall i \leq k < j, \langle \pi, k, \theta \rangle \models \neg B\bigg)$$
$$\lor$$
$$\neg\bigg(\Big[\forall j \geq i : \langle \pi, j, \theta \rangle \models \neg B \lor (\exists j \geq i : \langle \pi, j, \theta \rangle \models B \land \forall j \geq i, \theta' : \langle \pi, j, \theta'\theta \rangle \models \neg A)\Big] \land \forall i \leq k < j, \langle \pi, k, \theta \rangle \models \neg B\bigg)$$
$$\text{iff } \bigg(\neg\Big[\forall j \geq i : \langle \pi, j, \theta \rangle \models \neg B \lor (\exists j \geq i : \langle \pi, j, \theta \rangle \models B \land \forall j \geq i, \theta' : \langle \pi, j, \theta'\theta \rangle \models A)\Big] \lor \neg\forall i \leq k < j, \langle \pi, k, \theta \rangle \models \neg B\bigg)$$
$$\lor$$
$$\bigg(\neg\Big[\forall j \geq i : \langle \pi, j, \theta \rangle \models \neg B \lor (\exists j \geq i : \langle \pi, j, \theta \rangle \models B \land \forall j \geq i, \theta' : \langle \pi, j, \theta'\theta \rangle \models \neg A)\Big] \lor \neg\forall i \leq k < j, \langle \pi, k, \theta \rangle \models \neg B\bigg)$$
$$\text{iff } \bigg(\Big[\neg\forall j \geq i : \langle \pi, j, \theta \rangle \models \neg B \land \neg(\exists j \geq i : \langle \pi, j, \theta \rangle \models B \land \forall j \geq i, \theta' : \langle \pi, j, \theta'\theta \rangle \models A)\Big] \lor \exists i \leq k < j, \langle \pi, k, \theta \rangle \models B\bigg)$$
$$\lor$$
$$\bigg(\Big[\neg\forall j \geq i : \langle \pi, j, \theta \rangle \models \neg B \land \neg(\exists j \geq i : \langle \pi, j, \theta \rangle \models B \land \forall j \geq i, \theta' : \langle \pi, j, \theta'\theta \rangle \models \neg A)\Big] \lor \exists i \leq k < j, \langle \pi, k, \theta \rangle \models B\bigg)$$
$$\text{iff } \bigg(\Big[\exists j \geq i : \langle \pi, j, \theta \rangle \models B \land (\neg\exists j \geq i : \langle \pi, j, \theta \rangle \models B \lor \neg\forall j \geq i, \theta' : \langle \pi, j, \theta'\theta \rangle \models A)\Big] \lor \exists i \leq k < j, \langle \pi, k, \theta \rangle \models B\bigg)$$
$$\lor$$
$$\bigg(\Big[\exists j \geq i : \langle \pi, j, \theta \rangle \models B \land (\neg\exists j \geq i : \langle \pi, j, \theta \rangle \models B \lor \neg\forall j \geq i, \theta' : \langle \pi, j, \theta'\theta \rangle \models \neg A)\Big] \lor \exists i \leq k < j, \langle \pi, k, \theta \rangle \models B\bigg)$$

iff $\Big(\big[\exists j \geq i : \langle \pi, j, \theta \rangle \models B \land (\forall j \geq i : \langle \pi, j, \theta \rangle \models \neg B \lor \exists j \geq i, \theta' : \langle \pi, j, \theta'\theta \rangle \models \neg A)\big] \lor \exists i \leq k < j, \langle \pi, k, \theta \rangle \models B\Big)$
$\lor$
$\Big(\big[\exists j \geq i : \langle \pi, j, \theta \rangle \models B \land (\forall j \geq i : \langle \pi, j, \theta \rangle \models \neg B \lor \exists j \geq i, \theta' : \langle \pi, j, \theta'\theta \rangle \models A)\big] \lor \exists i \leq k < j, \langle \pi, k, \theta \rangle \models B\Big)$
iff $\exists i \leq k < j, \langle \pi, k, \theta \rangle \models B \lor \Bigg[\Big(\big[\exists j \geq i : \langle \pi, j, \theta \rangle \models B \land (\forall j \geq i : \langle \pi, j, \theta \rangle \models \neg B \lor \exists j \geq i, \theta' : \langle \pi, j, \theta'\theta \rangle \models \neg A)\big]\Big) \lor \Big(\big[\exists j \geq i : \langle \pi, j, \theta \rangle \models B \land (\forall j \geq i : \langle \pi, j, \theta \rangle \models \neg B \lor \exists j \geq i, \theta' : \langle \pi, j, \theta'\theta \rangle \models A)\big]\Big)\Bigg]$
iff $\exists i \leq k < j, \langle \pi, k, \theta \rangle \models B \lor \exists j \geq i : \langle \pi, j, \theta \rangle \models B \land \Bigg[\big[(\forall j \geq i : \langle \pi, j, \theta \rangle \models \neg B \lor \exists j \geq i, \theta' : \langle \pi, j, \theta'\theta \rangle \models \neg A)\big] \lor \big[(\forall j \geq i : \langle \pi, j, \theta \rangle \models \neg B \lor \exists j \geq i, \theta' : \langle \pi, j, \theta'\theta \rangle \models A)\big]\Bigg]$
iff $\exists i \leq k < j, \langle \pi, k, \theta \rangle \models B \lor (\exists j \geq i : \langle \pi, j, \theta \rangle \models B \land \forall j \geq i : \langle \pi, j, \theta \rangle \models \neg B)$
iff $\mathbf{F}(B) \lor \bot$
iff $\mathbf{F}(B)$

As seen earlier, it can be discussed whether this is a valid result.

C.3.4 Proof of K3

Axiom $K3$ states that $\mathcal{O}(A \land A'|B) \equiv (\mathcal{O}(A|B) \land \mathcal{O}(A'|B))$: DDL is closed under conjunction of the maintenance condition.

$\langle \pi, i, \theta \rangle \models \mathcal{O}(A \land A'|B)$
iff $\big[\forall j \geq i : \langle \pi, j, \theta \rangle \models \neg B \lor (\exists j \geq i : \langle \pi, j, \theta \rangle \models B \land \forall j \geq i, \theta' : \langle \pi, j, \theta'\theta \rangle \models (A \land A'))\big] \land \forall i \leq k < j, \langle \pi, k, \theta \rangle \models \neg B$
iff $\big[\forall j \geq i : \langle \pi, j, \theta \rangle \models \neg B \lor (\exists j \geq i : \langle \pi, j, \theta \rangle \models B \land \forall j \geq i, \theta' : \langle \pi, j, \theta'\theta \rangle \models A \land \forall j \geq i, \theta' : \langle \pi, j, \theta'\theta \rangle \models A')\big] \land \forall i \leq k < j, \langle \pi, k, \theta \rangle \models \neg B$
iff $\Big[\forall j \geq i : \langle \pi, j, \theta \rangle \models \neg B \lor \big[(\exists j \geq i : \langle \pi, j, \theta \rangle \models B \land \forall j \geq i, \theta' : \langle \pi, j, \theta'\theta \rangle \models A) \land (\exists j \geq i : \langle \pi, j, \theta \rangle \models B \land \forall j \geq i, \theta' : \langle \pi, j, \theta'\theta \rangle \models A')\big]\Big] \land \forall i \leq k < j, \langle \pi, k, \theta \rangle \models \neg B$
iff $\Big[\big[\forall j \geq i : \langle \pi, j, \theta \rangle \models \neg B \land (\exists j \geq i : \langle \pi, j, \theta \rangle \models B \land \forall j \geq i, \theta' : \langle \pi, j, \theta'\theta \rangle \models A)\big] \land \big[\forall j \geq i : \langle \pi, j, \theta \rangle \models \neg B \land (\exists j \geq i : \langle \pi, j, \theta \rangle \models B \land \forall j \geq i, \theta' : \langle \pi, j, \theta'\theta \rangle \models A')\big]\Big] \land \forall i \leq k < j, \langle \pi, k, \theta \rangle \models \neg B$
iff $\Big[\big[\forall j \geq i : \langle \pi, j, \theta \rangle \models \neg B \land (\exists j \geq i : \langle \pi, j, \theta \rangle \models B \land \forall j \geq i, \theta' : \langle \pi, j, \theta'\theta \rangle \models A)\big] \land \forall i \leq k < j, \langle \pi, k, \theta \rangle \models \neg B\Big] \land \Big[\big[\forall j \geq i : \langle \pi, j, \theta \rangle \models \neg B \land (\exists j \geq i : \langle \pi, j, \theta \rangle \models B \land \forall j \geq i, \theta' : \langle \pi, j, \theta'\theta \rangle \models A')\big] \land \forall i \leq k < j, \langle \pi, k, \theta \rangle \models \neg B\Big]$
iff $\langle \pi, i, \theta \rangle \models \mathcal{O}(A|B) \land \langle \pi, i, \theta \rangle \models \mathcal{O}(A'|B)$

This is a straightforward proof that is trivially achieved by the properties of FO-LTL (see Section 4.1).

C.3.5 Proof of K4

Axiom $K4$ states that $\mathcal{O}(A|B \vee B') \equiv (\mathcal{O}(A|B) \wedge \mathcal{O}(A|B'))$, that is, DDL is closed under disjunction of the activating condition.

$\langle \pi,i,\theta \rangle \models \mathcal{O}(A \wedge A'|B)$
iff $\Big[\forall j \geq i : \langle \pi,j,\theta \rangle \models \neg(B \vee B') \vee (\exists j \geq i : \langle \pi,j,\theta \rangle \models (B \vee B') \wedge \forall j \geq i,\theta' : \langle \pi,j,\theta'\theta \rangle \models A)\Big] \wedge \forall i \leq k < j, \langle \pi,k,\theta \rangle \models \neg(B \vee B')$
iff $\Big[\forall j \geq i : \langle \pi,j,\theta \rangle \models (\neg B \wedge \neg B')) \vee (\exists j \geq i : \langle \pi,j,\theta \rangle \models (B \vee B') \wedge \forall j \geq i,\theta' : \langle \pi,j,\theta'\theta \rangle \models A)\Big] \wedge \forall i \leq k < j, \langle \pi,k,\theta \rangle \models (\neg B \wedge \neg B')$
iff $\Big[(\forall j \geq i : \langle \pi,j,\theta \rangle \models \neg B \wedge \forall j \geq i : \langle \pi,j,\theta \rangle \models \neg B') \vee [(\exists j \geq i : \langle \pi,j,\theta \rangle \models B \vee \exists j \geq i : \langle \pi,j,\theta \rangle \models B'] \wedge \forall j \geq i,\theta' : \langle \pi,j,\theta'\theta \rangle \models A)\Big] \wedge (\forall i \leq k < j, \langle \pi,k,\theta \rangle \models \neg B \wedge \forall i \leq k < j, \langle \pi,k,\theta \rangle \models \neg B')$
iff $\Big[(\forall j \geq i : \langle \pi,j,\theta \rangle \models \neg B \wedge \forall j \geq i : \langle \pi,j,\theta \rangle \models \neg B') \vee [(\exists j \geq i : \langle \pi,j,\theta \rangle \models B \wedge \forall j \geq i,\theta' : \langle \pi,j,\theta'\theta \rangle \models A) \vee (\exists j \geq i : \langle \pi,j,\theta \rangle \models B' \wedge \forall j \geq i,\theta' : \langle \pi,j,\theta'\theta \rangle \models A)]\Big] \wedge (\forall i \leq k < j, \langle \pi,k,\theta \rangle \models \neg B \wedge \forall i \leq k < j, \langle \pi,k,\theta \rangle \models \neg B')$
iff $\Big[\Big(\forall j \geq i : \langle \pi,j,\theta \rangle \models \neg B \vee (\exists j \geq i : \langle \pi,j,\theta \rangle \models B \wedge \forall j \geq i,\theta' : \langle \pi,j,\theta'\theta \rangle \models A) \vee (\exists j \geq i : \langle \pi,j,\theta \rangle \models B' \wedge \forall j \geq i,\theta' : \langle \pi,j,\theta'\theta \rangle \models A)\Big) \wedge \Big(\forall j \geq i : \langle \pi,j,\theta \rangle \models \neg B' \vee (\exists j \geq i : \langle \pi,j,\theta \rangle \models B \wedge \forall j \geq i,\theta' : \langle \pi,j,\theta'\theta \rangle \models A) \vee (\exists j \geq i : \langle \pi,j,\theta \rangle \models B' \wedge \forall j \geq i,\theta' : \langle \pi,j,\theta'\theta \rangle \models A)\Big)\Big] \wedge (\forall i \leq k < j, \langle \pi,k,\theta \rangle \models \neg B \wedge \forall i \leq k < j, \langle \pi,k,\theta \rangle \models \neg B')$
iff $\Big[\Big(\forall j \geq i : \langle \pi,j,\theta \rangle \models \neg B \vee \forall j \geq i,\theta' : \langle \pi,j,\theta'\theta \rangle \models A \vee (\exists j \geq i : \langle \pi,j,\theta \rangle \models B' \wedge \forall j \geq i,\theta' : \langle \pi,j,\theta'\theta \rangle \models A)\Big) \wedge \Big(\forall j \geq i : \langle \pi,j,\theta \rangle \models \neg B' \vee (\exists j \geq i : \langle \pi,j,\theta \rangle \models B \wedge \forall j \geq i,\theta' : \langle \pi,j,\theta'\theta \rangle \models A) \vee \forall j \geq i,\theta' : \langle \pi,j,\theta'\theta \rangle \models A\Big)\Big] \wedge (\forall i \leq k < j, \langle \pi,k,\theta \rangle \models \neg B \wedge \forall i \leq k < j, \langle \pi,k,\theta \rangle \models \neg B')$
iff $\Big[\Big(\forall j \geq i : \langle \pi,j,\theta \rangle \models \neg B \vee \forall j \geq i,\theta' : \langle \pi,j,\theta'\theta \rangle \models A\Big) \wedge \Big(\forall j \geq i : \langle \pi,j,\theta \rangle \models \neg B' \vee \forall j \geq i,\theta' : \langle \pi,j,\theta'\theta \rangle \models A\Big)\Big] \wedge (\forall i \leq k < j, \langle \pi,k,\theta \rangle \models \neg B \wedge \forall i \leq k < j, \langle \pi,k,\theta \rangle \models \neg B')$
iff $\Big[\Big(\forall j \geq i : \langle \pi,j,\theta \rangle \models \neg B \vee \forall j \geq i,\theta' : \langle \pi,j,\theta'\theta \rangle \models A\Big) \wedge (\forall i \leq k < j, \langle \pi,k,\theta \rangle \models \neg B\Big] \wedge \Big[\Big(\forall j \geq i : \langle \pi,j,\theta \rangle \models \neg B' \vee \forall j \geq i,\theta' : \langle \pi,j,\theta'\theta \rangle \models A\Big) \wedge \forall i \leq k < j, \langle \pi,k,\theta \rangle \models \neg B')\Big]$
iff $\langle \pi,i,\theta \rangle \models \mathcal{O}(A|\ B) \wedge \langle \pi,i,\theta \rangle \models \mathcal{O}(A|\ B')$

Similarly to what happens with the previous axiom, this is also a trivial result based on the properties of FO-LTL.

Index of Terms

TLPLAN, 125, 130
2APL, 159–161

Action, 18, 125
Action Language, 19
ADL, 125, 126
Agency, 107
Autonomous Agent, 2, 12

BDI, 13, 92

Classical Planning, 18, 21
Constitutive Norm, 34
Counts-As Rule, 34

Derived predicate, 148
Dyadic Deontic Logic, 36, 109, 245

Electronic Institution, 30, 32
Enacts, 26, 82

Goal, 18

Hierarchical Task Network, 21

Institution, 25, 29

Linear Temporal Logic, 105

Metamodel, 69
Metric-FF, 157
Model-Driven Engineering (MDE), 69

Norm, 32, 84, 85, 108, 140
Norm Instance, 97, 110, 145
Norm Lifecycle, 97
Normative Model, 108
Normative Planner/Planning, 5, 127, 128
Normative Planning Problem, 127, 149

Organisation, 25–27

PDDL, 21, 125
Plan, 18, 22, 126
Planning, 18
Practical Reasoning, 12
Progression of Formula, 48, 130

Regulative Norm, 33
Repair Norm, 100
Role, 26, 82

Self-Loop Alternating Automaton, 111
Standard Deontic Logic, 36, 116
STRIPS, 21

Temporal Deontic Logic, 36