Charters

Nardine Osman

20 April 2012

Abstract

Norms have been used extensively in distributed open systems to regulate community behaviour. In this paper, we argue the need to distinguish between declarative norms, which are necessary for agent reasoning over the properties of a system, and operational norms, which are necessary for enforcing norms in a system in an efficient manner. We present a deontic-based policy language for specifying declarative norms, and we illustrate how we can verify the satisfaction of a given set of declarative norms against a given set of operational norms.

1 Introduction

The deployment of open distributed systems is increasing rapidly. The advances of network technologies are spawning a surge of application domains: ambient intelligence, cloud computing, service oriented computing, sensor networks, or virtual organisations, just to mention a few. These applications are composed of a wealth of physical devices, software components and, frequently, humans. The overall co-ordination of these elements presents a tremendous challenge because of: the lack of a centralised control, the openness of the system, and the complexity and volatility of the environment.

How are open distributed systems to be managed? Two choices are: either by some external managing agent or by engineering self-managing systems. Management by an external agent is complicated by the need to comprehend the changing requirements of the systems diverse components and is not scalable to large systems with thousands of millions or components. Self-management seems to be the only way (see the Autonomic Computing manifesto,\(^1\) for instance) and has to be achieved by building sufficient intelligence into the system. For distributed systems, multiagent technology is a natural choice as it emphasises the autonomy of its agents, and autonomy is the key to delivering the required self-management capability.

Our overall goal is thus to enable an open distributed system, composed of both humans and software agents, to manage itself. However, the evolution of such communities is the evolution of the norms that govern these communities. The study of norms and its effectiveness in regulating community behaviour has attracted a lot of attention in the field of multiagent systems. Some of the proposed approaches for defining norms has followed a declarative approach, where the rules specified the properties that need to be satisfied by a community. This is especially useful for reasoning about the system. Many others have followed an operational approach, where the norms are used for regulating behaviour by specifying who can perform what action and under what conditions. This is especially useful for coordinating multiagent interactions.

\(^1\)http://www.research.ibm.com/autonomic/manifesto/
Consider, for instance, the declarative norm that states that “public spaces should be smoke free”, and the operational norm that states that “smoking on a bus results in a €200 fine”. The operational rule is one way of enforcing the declarative one. We refer to declarative norms as charters, and the operational ones as bylaws. While the charter describes what the system should be, bylaws describe how the system should behave to satisfy the charter.

Both charters and bylaws are necessary. The need for bylaws is emphasised by the need for the specification of operational rules that help the execution of community interactions. The need for charters is emphasised by the need to reason about the properties and goals of a community. This is especially useful during the design phase of a community, as well as the evolution phase, where bylaws or charters may need to be revisited.

In this paper, we focus our study on charters by presenting a formal deontic-based policy language for specifying charters, while keeping in mind that these charters are designed for evolving communities where both humans and software agents interact. As such, they need to be simple and comprehensible for human users, yet formal to allow agent reasoning over them. We also study the relations between charters and bylaws, and illustrate how model checking may be used for the automatic verification of charters against bylaws whenever needed.

The remainder of this paper is divided as follows: Section 2 opens with a more elaborate introduction to charters by highlighting the differences between charters and bylaws; Section 3 discusses the literature review of the related work in the field of multiagent systems; Section 4 presents our proposed deontic-based policy language for specifying charters; Section 5 illustrates how the satisfaction of charters against bylaws may be formally verified via model checking; and finally, Section 6 concludes with a brief summary and a few words on future work.

2 Charters and Bylaws

According to the Oxford English dictionary, a charter is “a written evidence, instrument, or contract executed between man and man”, it is a document “granting privileges to, or recognising rights of, the people, or of certain classes or individuals”. We say a charter is essentially a convention, agreed upon by a set of participants, that sets the rules of the interaction between these participants by declaring the rights and privileges each participant has. Accordingly, participants are legally bound to obey and observe these rules, their contract.

In its most basic form, a charter $C$ may be composed of a set of rights $R$, where a right is defined as the tuple $R = \langle o, e, D \rangle$, and $o$ represents the object that the entity $e$ has a right to, and $D$ represents the set of conditions under which this right holds. For example, $\langle vote, citizen, is\_adult(citizen) \rangle$ states that a citizen has the right to vote if s/he is an adult.

However, we note that the rights of one participant usually imply the obligation or forbiddance of another participant with respect to performing some action(s). For example, the right to education implies an obligation on the government to provide education to their citizens. As such, deontic systems have been used to specify the “rights and duties” of individuals and organisations. In a deontic logic, a system is specified by a set of permissions, obligations, and related concepts. As this paper illustrates, we follow this approach and adopt the notion of deontic rules in our specification of charters.

In organisations and institutions, charters are used to specify the set of declarative rules (or norms). An example of such a declarative rule is that reviewers should provide their reviews promptly. However, in practice, the need arises for concrete details on who may perform what action, when to carry
out that action, and under what conditions (where the conditions are usually context dependent: different states of the interaction require different conditions). As we move from simple human interactions to more complex ones, enforcing the rules and guaranteeing an entity’s right becomes a challenge. To this end, bylaws are drafted under the authority of the charter. We think of bylaws as the set of operational rules, as opposed to declarative ones. For example, instead of simply stating that reviewers should provide their reviews promptly, the operational rule could state that after a reviewer receives a paper, they should submit their review in one month, at the latest. This becomes especially useful in distributed open systems that are composed of autonomous interacting agents, in which one common approach is to make use of a more precise plan of action to guide agent actions — as opposed to relying on a set of declarative rules. This plan of action is usually defined through a flow graph that essentially specifies the order in which the steps must be executed, which steps could be executed in parallel, etc. The entire flow graph should be consistent with the charter, and it can aid in penalising misbehaving participants.

Formally, a bylaw may then be defined as tuple: \( B = \langle S, R, A, T \rangle \), where \( S \) represents the set of states, \( R \) is the set of roles that users may play in the interaction, \( A \) is the set of actions that may be performed, and \( T : S \times A \times R \rightarrow S \) is a transition function that defines which role can perform which action at which state, and the new state resulting from that action. The tuple \( B \) is analogous to Kripke structures and finite state machines.

We note that the set of rights \( R \) specified by the charter \( C \) should map with the transitions of \( T \) (the transitions follow the rights \( R \)). As such, it is necessary to be able to verify that a specific bylaw satisfies its charter (\( B \models C \)). Key to this aspect of our research is the notion of self-regulated evolving communities. Automated verification is especially useful during the evolution (and construction) phase when community members need to discuss their bylaws and verify that their new suggested bylaws satisfy the charter (although sometimes the charters themselves may also need to be revisited). Formal verification mechanisms (such as theorem proving, SAT solvers, or model checking) may be used to prove such satisfaction properties, although we focus on model checking as it provides an automated verification mechanism, as illustrated by Section 5.

3 A Brief Literature Review

Norms are the rules that govern behaviour in groups and societies [1]. They essentially motivate and influence individual actions by dictating what values, beliefs, attitudes, and behaviours are deemed appropriate or not. Social norms have been extensively studied by anthropologists, sociologists, philosophers, and economists in the hope of understanding how they motivate individual actions, influence market behaviour, and so on. In multiagent systems, the study of norms gained tremendous attention due to the critical issue of coordinating agent behaviour and actions.\(^2\) The literature provides a variety of solutions that deal with specifying and regulating interactions in multiagent systems based on the concept of following social norms [26], such as having contracts and commitments [6], organisational approaches [14], electronic institutions [10], and distributed dialogues [23].

Existing approaches may be divided into two main categories: declarative approaches and operational ones. Declarative formalisms focus on the expressiveness of norms, the formal semantics, and how to resolve conflicts arising from an inconsistent set of norms. Declarative approaches are usually based on deontic logic, which is the logic of duties (which, as we illustrated earlier, deals with

\(^2\)We note that unlike other social sciences, the distinction between social and legal norms has not been concrete in the field of computer science.
concepts like permissions, prohibitions, and obligations, that help specify who can do what and under what conditions). Outside multiagent systems, deontic-based policy languages have been used widely in hardware systems and networks for security reasons, trust negotiation, access control, authorisation, and so on. [27] defines policies to be “one aspect of information which influences the behaviour of objects within the system”. [4] categorises policies into two types. The first type covers obligation policies for managing actions, which are usually event triggered condition-action rules. The basic concept is that specific events trigger specific actions, and the actions may only be executed if a predefined set of conditions is satisfied. The second type covers authorisation policies, which are usually used for access control.

In multiagent systems, several deontic based formal logics have been proposed for defining a normative specification of agents interaction, such as [21], [5], and [7]. In [21], it is proposed that a community should be defined whenever a group of agents have some common goals and they need to act as a whole within the society in order to fulfil them. Thus, they propose to define the roles within the community, the relationships among them, which actions each role can do, and how the obligations are distributed among the roles. Each role has associated deontic notions that describe the role obligations, permissions and prohibitions. [5] extends the BDI model of agents to include goals, obligations, and norms; the proposal is essentially based on providing a formal definitions of norms by means of some variation of deontic logic that includes conditional and temporal aspects. In order to verify that given interactions satisfy the declarative norms, [7] translates these declarative norms into concrete norms, which must be expressed in terms similar to those by which operational norms are specified, which we discuss next.

Although the declarative aspects of norms are important, operational norms are crucial to ensure that norms are operationally implemented. Among the operational approaches of multiagent systems are electronic institutions [10] and the lightweight coordination calculus [23]. In [10], it is argued that open multi-agent systems can be effectively designed and implemented as agent mediated electronic institutions where heterogeneous (human and software) agents can participate, playing different roles and interacting by means of speech acts. An institution is defined by a set of roles that agents participating in the institution will play, a common language to allow heterogenous agents to exchange knowledge, the valid interactions that agents may have structured in conversations, and the consequences of agents’ actions within an institution, captured by obligations that agents acquire and fulfil. These ideas have been validated through the realisation of an electronic auction house [24] and a conference centre [22]. In order to facilitate the institution designer’s work, a specification and verification tool for electronic institutions has been developed [9]. In [30, 29], a logical formalism for electronic institutions is presented as well as a simulation platform that permits the simulation of an institution by different agent populations.

The lightweight coordination calculus (LCC) [23] is a process calculus, based on logic programming, that provides means for achieving coordination in distributed systems by enforcing social norms. The process calculus specifies what actions agents can perform, when they can perform such actions, under what conditions these actions may be carried out, and so on. However, unlike electronic institutions, there are no “governors” than ensure that agents abide by norms. Of course, like all the approaches above, these rules are associated with roles rather than physical agents; and agents can play more than one role in more than one interaction. This provides an abstraction for the interaction model from the individual agents that might engage in such an interaction. Nevertheless, agents’ autonomy is preserved in the sense that it is up to the agents to decide: (1) whether or not to join an interaction; and (2) in which direction to drive the non-deterministic interaction models. The first case is dealt with by the agent’s decision making process, which is outside the scope of the LCC specifi-
cation. The second is achieved in LCC by making use of constraints whose satisfaction relies on the knowledge of the particular agent playing that role at that specific time. Unlike other process calculi, LCC is used both as a specification language to model interactions and as the executable model. Furthermore, LCC provides a lightweight language that has only two basic engineering requirements from agents: (1) to be able to extract the current state of the interaction and the agent’s next permissible action(s), and (2) to have an appropriate constraint solver for dealing with LCC constraints.

In this work, we try to bridge the gap between declarative and operational approaches by understanding the dynamics between the two. We propose a deontic-based policy language for the specification of charters, and we distinguish it from existing approaches by keeping it as simple as possible in such a way that it may also be comprehensible for human members of a community. Furthermore, we study how operational norms may be automatically verified against declarative ones.

4 Charter Specification: a Deontic-Based Policy Language (DPL)

This section presents our proposed deontic-based policy language (DPL) for specifying declarative norms. The section opens with a motivation behind our choice in designing DPL, followed by the syntax and semantics of the languages, and an example illustrating how declarative norms may be specified via DPL.

4.1 Motivation: Deontic Logic and Policy Languages

Deontic Logic. Deontic logic is the logic of duties. It deals with concepts like permissions, prohibitions, obligations, etc. It may be viewed as one way of defining social norms by specifying who can do what. In practice, research on deontic logic has focused on five main notions, defined through the following five operators: $P$ for permission, $O$ for obligation, $F$ for forbiddance or prohibition, $G$ for gratuitousness or what is non-obligatory, $I$ for indifference or what is optional. Usually, one of the operators is taken as a basic operator and the remaining four are defined in its term. In what follows, we define the operators in terms of the permission operator $P$ as follows: $Pa$ (action $a$ is permitted), $Oa \equiv \neg Pa$ (action $a$ is obligatory), $Fa \equiv \neg Pa$ (action $a$ is forbidden), $Ga \equiv P \neg a$ (action $a$ is gratuitous), and $Ia \equiv (Pa) \land (\neg Pa)$ (action $a$ is indifferent).

To achieve a better understanding of the deontic operators we rely on set theory. Actions can be divided into two main sets $P$ and $G$ describing what is permissible and what is gratuitous, respectively, as illustrated by Figure 1. The set of obligatory actions $O$ is then equivalent to the complement of $G$ in $P$. Similarly, the set of forbidden actions $F$ is equivalent to the complement of $P$ in $G$. Finally, the set of indifferent actions $I$ is essentially the intersection of sets $P$ and $G$.

![Deontic sets](image)

Figure 1: Deontic sets

There are various systems defining deontic logic: [18]'s deontic logic, [31, 32]'s dyadic deontic logic, [13]'s standard deontic logic, to name a few. Each of these specifies its own set of axioms and rules. We do not intend to provide a complete survey of deontic logic systems; however, to provide an insight of these systems, we introduce the semantics of the Standard Deontic Logic (SDL), the
“benchmark system of deontic logic”, as described by the Stanford Encyclopedia of Philosophy. The axioms and rules of SDL are presented below.

Axiom 1: All tautologous wffs of the language \((TAUT)\)

Axiom 2: \(O(p \rightarrow q) \rightarrow (Oq \rightarrow Oq)\) \((K)\)

Axiom 3: \(Op \rightarrow \neg Op\) \((D)\)

Rule 1: \(\frac{p, p \rightarrow q}{q} \) \((MP)\)

Rule 2: \(\frac{q}{Op} \) \((O-NEC)\)

The first axiom \((TAUT)\) is common to all logics. The second \((K)\) is a well known property of modal logics in general. In the context of deontic logic, it specifies that if there is an obligation to \(p \rightarrow q\) and there is an obligation to \(p\), then there is an obligation to \(q\). The third axiom is called \((D)\) for deontic;\(^3\) it states that if there is an obligation to \(p\) then \(p\) must be permitted, i.e. there is no obligation for \(\neg p\). The inference rules of this logic are the propositional calculus modus ponens rule \((MP)\) and the obligation necessitation rule \((O-NEC)\), which states that if \(p\) holds then the obligation to \(p\) also holds.

Several theorems and rules may be derived from the SDL system. For instance, the following rule \(\frac{p \rightarrow q}{Oq \rightarrow Oq} \), named \((O-RM)\), may be derived as follows:

\[
\frac{p \rightarrow q}{O(p \rightarrow q)} \quad (O-NEC) \\
\frac{Op \rightarrow Oq}{Op} \quad (K)
\]

Applying the same proof to the tautology \(p \rightarrow p \lor q\) results in the corollary \(Op \rightarrow Op(p \lor q)\). However, [25] identifies a paradox resulting from this corollary. He points out that this corollary results in statements such as “If I ought to mail a letter, then I ought to mail or burn it”, which is clearly unacceptable.

The paradox presented above is only one of the numerous paradoxes of deontic logic. Deontic logic systems are famous for the paradoxes they raise. However, we do not dwell on these, since this research work does not make use of a deontic logic system, but focuses on the general concepts and ideas of deontic logic.

**Policy Languages.** Policy languages have been used widely in hardware systems and networks for security reasons, trust negotiation, access control, authorisation, to name a few. [28] defines policies to be “one aspect of information which influences the behaviour of objects within the system”. [3] categorises policies into two types. The first is the obligation policies for managing actions. These are usually event triggered condition-action rules. The basic concept is that specific events trigger specific actions, and the actions may only be executed if a predefined set of conditions is satisfied. The second type is the authorisation policies, which are usually used for access control. Deontic concepts that are based on permissions and prohibitions are usually used here. Since it is not practical to define policies relating to individual agents, policies are defined in the context of roles and organisational groups. For this reason, policy languages may be viewed as yet another method for the specification of social norms.

Policy languages have been defined for dealing with a variety of issues. For example, the ASL policy language [15] is strictly a security policy. Ponder [8] addresses both security and management issues. The Rei policy language [16] is a general purpose policy language that supports security issues.

\(^3\)Standard deontic logic SDL is sometimes simply referred to as system \(KD\) or even system \(D\), in reference to its axioms.
management, conversations, amongst others. Policy languages, like any other programming language, may be logic-based (e.g., ASL [15], Rei’s Prolog implementation [16], RDL [11]), object-oriented (e.g., Ponder [8], RuleML [2]), based on markup languages (e.g., Rei’s RDF implementation [16], XACML [19], TPL [12]), and so one.

The literature contains a huge collection of policy languages with different colours and flavours. Nevertheless, all policies are essentially (in their simplified form) a tuple of the form \((S, O, \langle \text{Sign} \rangle A)\) which permits or prohibits — depending on the sign of \(A\) — a subject \(S\) from executing an action \(A\) on an object \(O\). Obligations are usually even-triggered rules. Additionally, conditions may be attached to these rules. Available policy languages are basically more specialised versions of the above, each with its own conflict resolution mechanism.

One important issue in designing policy languages is that of conflict resolution. Usually conflicts between policies may arise. Different policy languages then propose different solutions, which is usually context based. For instance, one solution would be to explicitly define precedence rules. Another is to rely on more general rules, such as giving a negative authorisation a precedence over positive authorisations. Meta policies may also be used to test for conflicts and define application specific precedence rules (refer, for instance, to [8] for further information on Ponder and its meta policies).

Our proposed language for the specification of charters is similar in spirit to policy languages, although we do not yet provide mechanisms for addressing conflict resolution, as this is outside the scope of this work.

4.1.1 DPL Syntax and Semantics

Syntax. We introduce a deontic-based policy language for specifying the five deontic operators \(\mathcal{P}\), \(\mathcal{O}\), \(\mathcal{F}\), \(\mathcal{G}\), and \(\mathcal{I}\). Figure 2 presents the syntax of a DPL rule \(\Phi\).

\[
\Phi := \text{must}(\text{event}, s) \langle \leftarrow c \rangle \mid \text{can}(\text{event}, s) \langle \leftarrow c \rangle
\]

Figure 2: DPL syntax

The syntax states that a deontic rule is specified by one of the following predicates: \text{must}(\text{event}, s) or \text{can}(\text{event}, s), where \(s \in \{+, -\}\). The \text{event} specifies an event that may occur. It may either describe the action \text{ac} performed by a specific agent \text{ag} \((\text{action}(\text{ac}, \text{ag}))\) or it may describe an event that simply occurs in the environment and that is not brought about by a given agent \((\text{action}(\text{ac}))\). The latter type of event may be used to specify events such as ‘the sun is shining’ or ‘it is raining’. Finally, we note that the agent is usually defined as \text{agent}(\text{role}, \text{id})\), where \(\text{id}\) represents the agent’s unique identifier, and \text{role} represents the role an agent can play in a given system, such as \text{auctioneer} or \text{bidder} in an auction scenario. This separation between roles and agent identifiers allows the specification of generic deontic rules that do not apply to specific agents but to specific roles played by the agents.

Note that negative permissions, which represent both obligations \(\neg \mathcal{P} a\) and forbiddance \(\neg \mathcal{P} a\), are specified via the \text{must} DPL rule. For example, if an agent \text{ag} is obliged to perform action \text{ac}, then we say: \text{must}(\text{action}(\text{ac}, \text{ag}), +).\) However, if the agent is forbidden to perform \text{ac}, then we say: \text{must}(\text{agent}(\text{ac}, \text{ag}), -).
On the other hand, positive permissions, which are either normal permissions $\mathcal{P}a$ or gratuitousness $\mathcal{P}\neg a$, are specified via the $\textit{can}$ DPL rule. For example, if an agent $ag$ is permitted to perform the action $ac$ in a given interaction model, then we say: $\textit{can}(\text{action}(ac,ag),+)$. However, if the agent is gratuitous towards performing $ac$, then we say: $\textit{can}(\text{action}(ac,ag),-)$. As for indifference, it is defined as the conjunction of gratuitousness and permissions, i.e. a conjunction of negative and positive $\textit{can}$ predicates.

Additionally, conditions may be attached to deontic rules. The square brackets around $[\leftarrow c]$ imply that zero or one occurrence of this term is permitted. Conditions need to be specified by the language of the chosen model checker. Furthermore, they may also contain temporal properties, also to be specified in the temporal logic of the model checker. The addition of temporal properties to the conditions increases the richness of the properties that may be verified. For instance, one rule may state that an agent can make a payment under the condition that interaction guarantees that a receipt will eventually be sent. However, we note that the use of a temporal logic usually makes the conditions much harder to comprehend by human users. In such cases, syntactic sugaring will strongly be required. In our specific implementation, conditions are usually composed of Prolog terms, and if they contain temporal properties then these are specified in the $\mu$-calculus syntax [17]. While the DPL language itself is simple and comprehensible to a human user, future work will need to address the comprehensibility of DPL conditions specified in Prolog and the $\mu$-calculus.

**Semantics.** Deontic logic may be viewed as a special type of modal logic. An act $a$ that is obligatory can be defined in terms of the modal logic necessity operator $\square a$. What is permissible, or possible, can be defined in terms of the modal logic possibility operator $\Diamond a$, which is equivalent to $\neg \square \neg a$. What is forbidden is viewed as that act whose negation is obligatory, i.e. it may be defined in terms of $\square \neg a$. What is gratuitous is that which is not obligatory, i.e. it is defined as $\neg \square a$. Finally, what is indifferent is both possible and gratuitous, i.e. it is equivalent to $\neg \square \neg a \land \neg \square a$.

What is needed then is to interpret the deontic operators in the context of an operational interaction model, specified via some state-space defining the possible worlds. For example, instead of talking about permissions in general, we state the actions a particular agent is permitted to do with respect to a particular interaction. The state-space, specified via operational norms, is a representation of the various worlds that may be realised. Each path in the state-space represents a possible world that may unfold. We define the relation of the deontic operators to the interaction model’s state-space as follows.

$\textit{must}(event,+)$, which describes obligations, states that an event is obligatory if and only if the event is realisable for in all worlds/paths. Similarly, $\textit{must}(event,\neg)$, which describes forbiddance, states that an event is forbidden if and only if the event is not realisable in all worlds/paths. $\textit{can}(event,+)$, which describes permission, states that an event is permitted if and only if there exists at least one world/path where the event is realisable. $\textit{can}(event,\neg)$, which describes gratuitousness, states that an event is gratuitous if and only if there exists at least one world/path where the event is not realisable. Finally, we can deduce that an event is indifferent, which is specified as the conjunction of permission and gratuitousness ($\textit{can}(event,+)$ $\land$ $\textit{can}(event,\neg)$), if and only if there exists a world/path where the event is realisable and there also exists a world/path where the event is not realisable.

The semantics are formally defined by Figure 3, where $\langle B,s_0 \rangle \models \Phi$ states that the DPL rule $\Phi$ is satisfied at state $s_0$ of the transition model (or Kripke structure) $B$, and $\langle s_0 \rightarrow s_1 \rightarrow \ldots \rangle$ describes a path in the model $B$ that starts with the state $s_0$. 


\[ \langle B, s_0 \rangle \models \text{must}(\text{event}, +) \iff \forall \langle s_0 \rightarrow s_1 \rightarrow \ldots \rangle \cdot (\exists i \cdot \langle B, s_i \rangle \models \text{event}) \]
\[ \langle B, s_0 \rangle \models \text{must}(\text{event}, -) \iff \forall \langle s_0 \rightarrow s_1 \rightarrow \ldots \rangle \cdot (\forall i \cdot \langle B, s_i \rangle \not\models \text{event}) \]
\[ \langle B, s_0 \rangle \models \text{can}(\text{event}, +) \iff \exists \langle s_0 \rightarrow s_1 \rightarrow \ldots \rangle \cdot (\exists i \cdot \langle B, s_i \rangle \models \text{event}) \]
\[ \langle B, s_0 \rangle \models \text{can}(\text{event}, -) \iff \exists \langle s_0 \rightarrow s_1 \rightarrow \ldots \rangle \cdot (\forall i \cdot \langle B, s_i \rangle \not\models \text{event}) \]

Figure 3: DPL semantics

Figure 4 provides a graphical representation of the relation between the deontic sets and the occurrence of the deontically constrained actions in an interaction’s state-space. Note that if an action is realisable in a path, then the action may occur at some node in this path. However, if an action is not realisable in a path, then the action should not occur at any node in this path. Also note that the deontically constrained actions in an interaction’s state-space can now be specified in terms of general (CTL) temporal operators: \( A, G, E, \) and \( F \) (Figure 4).

\[ \begin{align*}
\mathcal{P} a & \quad O a \\
\text{can}(a, +) & \quad \text{must}(a, +) \\
EFa & \quad AFa \\
\end{align*} \]
\[ \begin{align*}
\mathcal{F} a & \quad \mathcal{G} a \\
\text{must}(a, -) & \quad \text{can}(a, -) \\
AG\neg a & \quad EG\neg a \\
\end{align*} \]
\[ \begin{align*}
\mathcal{I} a & \\
\text{can}(a, +) \land \text{can}(a, -) & \\
EFa \land EG\neg a & \\
\end{align*} \]

Figure 4: Mapping deontic operators into temporal ones

5 Verification of Charters against Bylaws

Differentiating between charters and bylaws raises the question of how can one maintain that bylaws do not violate the charter. We note that the set of rights \( R \) specified by the charter \( C \) should map with the transitions \( T \) of a bylaw \( B \). In other words, the transitions \( T \) should follow the rights \( R \). As such, it is necessary to be able to verify that a specific bylaw satisfies its charter: \( B \models C \). Additionally, the need for verification arises when agents need to join already existing communities, or when they argue and negotiate on how their community charter will evolve. In such cases, it is crucial for the agent to verify that its own specification is compatible with that of its community’s charters and bylaws.

As such, we say there are two objectives to the verification of charters against bylaws: (1) to be able to verify that a specific bylaw satisfies its charter; and (2) to be able to verify that a given agent’s specification is compatible with a given community. Automated verification is crucial here, since agents will want to verify their compatibility with a given community at runtime; similarly, the automated evolution phase of charters and bylaws requires an automated verification that bylaws do not violate their charters. We choose model checking from amongst other verification techniques
because it can provide a fully automatic verification process which could be carried out by the agents at interaction time. In particular, we use the model checker of [20] which can automatically verify the declarative DPL norms against operational ones specified in the lightweight coordination calculus (LCC).

The model checking problem is generally defined as follows: Given a system $B$ and a formula $c$, does $B$ satisfy $c$? In our case, the system is defined through the set of operational norms, and the formula is an element $c$ of a charter $C$. A charter $C$ is satisfied by a bylaw $B (B \models C)$, if and only if all the elements of the charter are satisfied by $B$: $B \models C \iff \forall c \in C : B \models c$. The bylaw $B$ to be verified must be modelled in the language of the model checker. This is a language, often a process calculus, that presents the system as a finite state transition system. A finite state transition system\(^4\) is usually presented via a state transition graph (or simply a state graph), and is a type of a non-deterministic finite state machine. We choose the model checker of [20], which specifies the bylaws $B$ in the lightweight coordination calculus (LCC). LCC is essentially a process calculus that is based on logic programming and created for the purpose of both specifying and executing interaction models. This makes LCC a model checker’s ideal candidate for runtime verification. Having the executable model fed directly to the model checker avoids the complexity of modelling the system in another language and the possibility of introducing errors in doing so. Furthermore, with LCC, agents are capable of automatically extracting the current state of the interaction and directly feeding it to the model checker when necessary. As for the specification of the charter $C$, these properties have traditionally been specified via some temporal (or modal) logic. Temporal properties distinguish themselves from other logics by introducing temporal features to the properties describing the behaviour of a system. Safety and liveness properties can then be specified by using temporal operators of the form it will never be true, it will eventually be true, etc. The chosen model checker uses the $\mu$-calculus temporal logic for the specification of $C$. We then provide a translation from DPL to the $\mu$-calculus, which we present next.

Translating DPL Rules into the $\mu$-Calculus. Deontic rules are specified via the DPL language of Figure 2. However, to verify whether these constraints will be broken or not, the verifier needs to study the occurrence of the actions constrained deontically in the given state-space. Since we are using model checking as our method of verification, then the occurrence of these actions needs to be specified in a temporal logic. Figure 4 has already provided a mapping between deontic sets and the occurrence of the deontically constrained action in an interaction’s state-space. This mapping has also associated DPL with the general temporal operators $A$, $G$, $E$, and $F$.

Translating DPL predicates into our chosen model checker’s temporal language, the $\mu$-calculus, is then a straightforward task, as the temporal semantics which were presented by Figure 3 simply need to be mapped to the semantics of the $\mu$-calculus [20]. Figure 5 presents the rewrite rules that provide this translation.

We do not present the syntax and semantics of the $\mu$-calculus in this paper, and we refer the reader interested in reading more about this topic to [17]. However, in what follows, we briefly explain the meaning of the $\mu$-calculus formulae of Figure 5.

Permission in the $\mu$-calculus is specified as follows: $\mu X. (\langle \text{event} \rangle \tt \lor \langle - \rangle X)$, which states that either the event can occur ($\langle \text{event} \rangle \tt$) or something can happen after which the property should be satisfied again ($\langle - \rangle X$). Finally, termination is guaranteed by the least fixpoint operator ($\mu X$). The guaranteed termination implies that the event will eventually occur in some path.

---

\(^4\)Model checking usually requires transition systems to be finite.
\[
must(event,+) \leftarrow c \quad \longrightarrow \quad (c \land \mu X.[-event]X \land \langle -\rangle \tt) \lor \ \\
(-c \land \nu X.\langle -event\rangle X \lor [-]\ff)
\]
\[
must(event,-) \leftarrow c \quad \longrightarrow \quad (c \land \nu X.[event]\ff \land [-]X) \lor \ \\
(-c \land \mu X.\langle event\rangle \tt \lor \langle -\rangle X)
\]
\[
can(event,+ \leftarrow c \quad \longrightarrow \quad (c \land \mu X.\langle event\rangle \tt \lor \langle -\rangle X) \lor \ \\
(-c \land \nu X.[event]\ff \land [-]X)
\]
\[
can(event,+ \leftarrow c \quad \longrightarrow \quad (c \land \nu X.\langle -event\rangle X \lor [-]\ff) \lor \ \\
(-c \land \mu X.[-event]X \land \langle -\rangle \tt)
\]

Figure 5: Rewrite rules translating DPL predicates into \(\mu\)-calculus formulae

\textit{Gratuitousness} in the \(\mu\)-calculus is specified as follows: \(\nu X.\langle -\text{event}\rangle X \lor [-]\ff\), which states that either the event in question cannot happen in at least one of the next steps, after which the property should be satisfied again (\(\langle -\text{event}\rangle X\)), or nothing can happen anymore (\([-]\ff\)). This property is satisfied infinitely often (\(\nu X\)). In summary, it states that there exists a path in which the event never happens.

\textit{Obligation} in the \(\mu\)-calculus is specified as follows: \(\mu X.[-\text{event}]X \land \langle -\rangle \tt\), which states that something can happen (\(\langle -\rangle \tt\)), and for all events that occur that are different from ‘event’, the same property should be satisfied again (\([\langle -\text{event}\rangle\]X\)). Finally, \(\mu X\) guarantees termination, implying that ‘event’ will eventually occur in all paths.

\textit{Forbiddance} in the \(\mu\)-calculus is specified as follows: \(\nu X.[\text{event}]\ff \land [-]X\), which states that the event is not permitted (\([\text{event}]\ff\)), and for every other event that occurs, the same property will be satisfied again (\([-]X\)) infinitely often (\(\nu X\)).

Furthermore, as illustrated earlier by Section 4.1.1 and our translation of Figure 5, if the condition of the deontic rule is satisfied, then \textit{can} predicates with a positive sign are treated as permissions, \textit{can} predicates with a negative sign as gratuitousness, \textit{must} predicates with a positive sign as obligations, and \textit{must} predicates with a negative sign as forbiddance.

But what if the condition of a DPL deontic rule is not satisfied? In this case, different interpretations may be accepted. For example, if the condition of a permission rule is not satisfied, do we assume this implies that the rule should not be satisfied? Or are we then indifferent towards the rule’s satisfaction? In our current implementation, as illustrated by Figure 5, every time a condition is not satisfied, the negation of the \(\mu\)-calculus specification should be satisfied. For example, permissible actions whose conditions have not been satisfied are treated as forbidden actions, and vice versa. Similarly, obligatory whose conditions have not been satisfied are treated as gratuitous actions, and vice versa.

\textbf{Model Checking Charters.} Given the rewrite rules of Figure 5, agents (whether software agents or human agents) can now call the model checker with specific DPL rules along with a specific implementation, or a bylaw, and automatically get an answer on whether the DPL rules are satisfied by the given bylaw or not. The bylaws fed to the model checker are specified in LCC, which is one P2P implementation of multiagent interactions that follows the spirit of electronic institutions. Future work will focus on the light version of P2P implementation of electronic institutions (EI-Lite) that is
currently being developed for this project.

6 Conclusion and Future Work

This paper has presented a logic (DPL) for the specification of declarative norms, or charters. The logic is simple and comprehensible to human users, and we have illustrated how it may be verified against operational norms in an automated and efficient manner via a logic-based model checker. This allows agents to verify at runtime that a specific bylaw satisfies its charter, and it may also be used to allow them to verify that their specification is compatible with a given community.

Future work will focus on three main issues: (1) the DPL language needs to be elaborated further to capture a richer set of properties, yet maintaining its current simple representation; (2) provide syntactic sugaring for the specification of the conditions of DPL rules, which will allow conditions to be comprehensible to human users; and (3) provide a translation that allows the model checker to use the EI-Lite system for the specification of operational norms.

References


