Simulating Normative Behaviour in Multi-agent Environments Using Monitoring Artefacts

Stephan Chang and Felipe Meneguzzi(✉)

School of Computer Science, Pontifical Catholic University of Rio Grande do Sul, Porto Alegre, Brazil
stephan.chang@acad.pucrs.br, felipe.meneguzzi@pucrs.br

Abstract. Norms are an efficient way of controlling the behaviour of agents while still allowing agent autonomy. While there are tools for programming Multi-Agent Systems, few provide an explicit mechanism for simulating norm-based behaviour using a variety of normative representations. In this paper, we develop an artefact-based mechanism for norm processing, monitoring and enforcement and show its implementation as a framework built with CArtAgO. Our framework is then empirically demonstrated using a variety of enforcement settings.

1 Introduction

Multi-Agent Systems are often used as a tool for simulating interactions between intelligent entities within societies, organisations or other communities. This Agent-based Simulation is useful for studying social behaviour in hypothetical situations or situations that may not be easily reproduced in the real world. The entities being simulated, human or otherwise, are represented by programmable intelligent agents, which must present reactive, pro-active and social behaviour [1].

When working with social simulations, we must consider that agents should be free to act in their own best interest, even though their actions might produce negative effects to other agents. For this reason, we establish rules that (1) prohibit actions that harm the society's performance; (2) oblige actions that maintain the society's well being; and (3) permit actions that can be beneficial to society, but never harmful. These rules, referred to as "norms" in multi-agent environments, allow agents to reason and act freely, while still being subject to punishment in the event that a norm is violated [2]. Although the purpose of norms is to mediate the interactions of agents in an environment, sometimes violating a norm can prove advantageous for an agent due to the reward of violation compensating for the penalties of detection. Existing work on normative reasoning [3–10] try to explore the trade-offs between compliance and non-compliance and propose new ways in which agents see and reason about norms. Still, there is no available tool that simulates norm-based behaviour to serve as a common ground for benchmarking implementations of normative behaviour and reasoning. In norm-based behaviour simulations we must define data structures for the various types of norms, including at least one of prohibitions, permissions or

V. Dignum et al. (Eds.): COIN 2015, LNAI 9628, pp. 59–77, 2016.
DOI: 10.1007/978-3-319-42691-4_4

obligations. Once these norms are active, agent interactions shall be observed by a monitoring mechanism and analysed by a norm-enforcing agent, which will then punish agents caught violating norms.

Although there are multiple frameworks that can be used to simulate agent societies, such as the MASSim [11] simulators, or the agent programming languages Jason [12] and JADE [13], relatively less attention has been focused on frameworks for norm-based behaviour simulation [14, Chap. 1]. In this paper, we bridge this gap by developing a scalable norm processing mechanism that performs monitoring and enforcement in multi-agent environments. Our contributions are a mechanism to monitor agents actions in an environment, described in Sect. 4.5 and a mechanism for norm maintenance and enforcement, described in Sect. 4.6. In Sect. 5 we demonstrate the functionality of our mechanism using an empirical experiment applying our mechanism to a Multi-Agent System.

2 Simulating Multi-agent Societies

When self-interested intelligent agents [1] share an environment, competition between them becomes inevitable [15]. This idea becomes clear when we think of multi-agent systems as societies. Each person in a society has their own goals and plans to achieve them, and it is in their best interest to do so by spending as little effort as possible. Take for an example a person interested in eating an apple and another interested in selling one. For the buying person, its goal is to acquire the apple from the seller for the lowest cost possible, preferably with no cost at all. For the seller, the goal is to sell the apple for as high a price as affordable by the buyer, maybe even higher than that. Now, considering that in this hypothetical world no notion of ethics is known yet, the buyer soon realizes that instead of paying for the apple he wants to eat, he could simply grab it and eat it on the spot.

Competition between agents is often intended when working with agent-based simulations, as we desire to see how agents perform under such circumstances. However, to prevent the system as a whole from descending into chaos, we must establish rules in order to control agent interactions while still allowing them to be autonomous. Nevertheless these rules must be limited to directing agents, rather than restraining them, otherwise, much of the benefit from autonomous agents is lost. When rules are set, agents that disregard them are subject to punishment for potentially harming the environment. In our buyer/seller system, we could establish a rule that guarantees items sold at shops must be paid for. If one is caught stealing, it will need to pay for the seller's injury. By doing so, we allow the buyer to reason about the advantages and disadvantages of obeying rules, letting it decide on an appropriate action plan. In multi-agent systems, we refer to these rules as norms.

Usual mechanisms for controlling agent interactions include interaction models, used by simulators such as NetLogo [16], MASON [17] and Repast [18]; strategies, commonly used in Game Theory; and organisation-oriented normative systems, such as $\mathcal{M}$oise [19]. The disadvantage of these methodologies is that agents are constrained to the rules of their environment. They are not allowed to break rules because the system is rule-compliant by design, also known as the regimentation approach [20]. However, unlike environmental constraints, perfect enforcement (regimentation) of social norms is unrealistic and undesirable, because it prevents occasional violations that would bring about a greater good [6,21].

3 Normative Scenario - Immigration Agents

To facilitate explanation and exemplification of our approach, as well as to highlight its capabilities, we present the scenario we use to test our mechanism. This scenario helps understand what norms are and how they control interactions in an environment. First, we present a short story that connects the environment to its agents, then we outline the norms that constrain them.

The government of a fictional emerging nation[1] started an immigration program to accelerate development through the hiring of foreigners. The country welcomes visitors, besides landed immigrants, to the country, since money from tourism greatly boosts the local economy. At the border, immigration officers must inspect immigrant passports. The foreigner acceptance policy is quite straightforward, and immigration agents must immediately accept immigrants with valid passports and no criminal records, and reject John Does and refugees outright. The government believes that the more immigrants it accepts, the better. Each officer's responsibility is to accept as many immigrants as possible, while still following the guidelines that were passed to them. Each accepted able worker nets the officer 5 credits, which eventually turn into a bonus to the officer's salary. There are no rewards for rejecting immigrants. It becomes clear that the bonus each officer accumulates depends entirely on chance, and some officers may accumulate more than others, if at all. As such, some officers might feel inclined to accept immigrants they should not, only to add to their personal gain.

To ensure officers act on the best interests of the nation only, the government introduced an enforcement system to the offices at the borders. Among the officers working in the immigration office, one is responsible for observing and recording the behaviour of those working in booths. This officer is known as the "monitor". His job is to write reports about what the officers do and send these reports to another officer, known as the "enforcer". The enforcer then reads the reports that are passed to him and look for any inconsistencies, such as the approval of an illegal immigrant. As this represents a violation of a rule, or norm, the enforcer then carries out an action to sanction the offending officer. The penalties for approving an illegal immigrant are the immediate loss of 10 credits and suspension of work activities for up to 10 s. Considering that immigrants arrive at a rate of 1 per 2 s, in a 10-s timespan 5 immigrants would have arrived at a given booth, meaning that a violating officer potentially loses 25 credits. Added to the other portion of the sanction, the potential loss rises up to 35 credits.

[1] Inspired by the game "Papers, please": http://papersplea.se.

The enforcement system, however, is not cost free. Each monitor and enforcer has an associated cost and it is within the interests of the nation to spend as little as possible with such a system. Therefore, the government wants to know how intensive the system must be to cover enough cases of disobedience so that officers will know violating norms is a disadvantage rather than an advantage.

There are two norms that can be extracted from this scenario, which we define in Examples 1 and 2. Later, in Sect. 4.3, we develop the formal representation of norms in our system and proceed to formally defining these norms. These norms concern the stability of the immigration program by assuring valid immigrants are accepted and discouraging corrupt officers to accept those who should not be.

Example 1. "All immigrants holding valid passports must be accepted. Failure to comply may result in the loss of 5 credits."

Example 2. "All immigrants holding passports that are not valid must not be accepted. Failure to comply may result in the loss of 10 credits and suspension from work activities for up to 10 s."

4 NormMAS Framework

In this section, we develop our monitoring and enforcement framework for normative agents. We start with an outline of the main components in our framework in Sect. 4.1. We them review the agent and environment-based approaches we use in our implementation in Sect. 4.2. Sections 4.3 and 4.4 describe the formalisation of norms and actions we adopt. With these formalisation covered, we explain how monitoring and enforcement work in Subsects. 4.5 and 4.6, respectively.

4.1 Architecture Overview

To allow the reader to better understand this section, we first offer an overview of the architecture envisioned by our work. We illustrate this architecture in Fig. 1, which shows the main elements that compose our framework and their interactions. These elements can be divided into three groups: *agents*, *environment* and *external*.

The *agents* group is self-explanatory, and it is where we put the agents that we are using for simulation and for monitoring/enforcement tasks. The "Simulation Agent Programs" are the agent programs which are simulating the behaviour we wish to study, in this case our immigration officers. "Monitor Agents" are agent programs which observe the actions performed by the simulation agent programs and "Enforcer Agents" make the decision of whether these actions violate some norm or not.

The *environment* group is composed of the elements that define what an environment is like. In our case, our environment is not a centralised entity, but a collection of artefacts through which agents interact. For example, monitor agents use the "Reporting Interface Artefact" to file reports for enforcer agents to analyse, as if they were actually putting reports in a pile over the enforcer agent's desk. As we describe in the next subsection, this approach makes programming the environment easier by separating responsibilities among different artefacts, instead of concentrating actions in a single environment description. The types of artefacts in this group should include all types pertaining to the simulation context, *e.g.* immigration booths for passport reviewing; and three fixed types that are part of our framework: the *reporting artefact*, the *monitoring artefact* and the *normative artefact*. These artefacts are used exclusively by monitoring and enforcement agents to perform tasks of the normative context, the exception being the Normative Artefact, which should be accessible to agents interested in observing normative events. Normative events include the creation, activation, deactivation and destruction of norms and the emission of sanctions to violating agents.

The *external* group is where we keep the elements that are auxiliary to our framework, and although not considered autonomous agents are also not part of the environment. Currently, this group contains the Action History, a structure in which we store actions for normative analysis, and the Normative Base, a database of established norms. In the following subsections we discuss each of these groups in more detail.

4.2 Jason and CArtAgO

In order to show the feasibility of the mechanism proposed in this paper, we use two programming approaches: agent-oriented programming and environment-oriented programming. The former is provided by the Jason interpreter [12], while the latter is achieved with the Common Artifact infrastructure for Agents Open environments (CArtAgO) [22].

Jason provides us with the means to program agents using the *AgentSpeak* language [23] in a Java environment. Agents are built with the BDI [24] architecture, and so their behaviour is directed by beliefs, goals and plans. Beliefs are logical predicates that represent an agent's considerations towards its environment. Predicates such as `valid(Passport)` and `wallet(50,dollars)` indicate that the agent believes the given passport variable is valid and that his wallet currently contains 50 dollars. In *AgentSpeak* variables start with an upper-case letter, while constants start with lower-case.

Goals are states that the agent desires to fulfil, and these can be either *achievement goals* or *test goals*. Achievement goals are objectives or milestones that agents pursue when carrying out their duties. To represent these in *AgentSpeak*, the goal's name is preceded by the '!' character. Test goals are questions an agent may ask about the current state of the environment. These can be identified by a '?' preceding the goal's name.

To achieve these goals, agents need to perform sequences of actions that modify the environment towards the desired states. This sequence of actions is referred to as a *plan* [25]. A plan is not necessarily composed solely of actions, however, it can also contain sub-plans. This allows complex behaviours to be built, creating flows of actions that vary and are influenced by agent beliefs and perceptions.

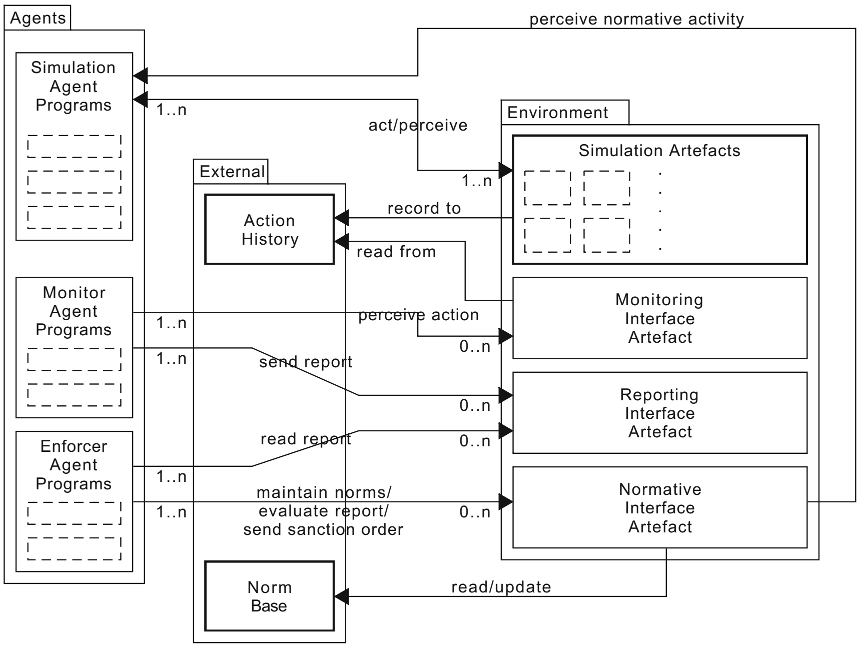

Fig. 1. Components of the NormMAS framework and their interactions.

As with any other programmed system, multi-agent systems must be tested before being effectively deployed to their end environments. To do so, test environments can be programmed for agents to be observed and any faulty behaviour addressed before release. Jason allows the programming of test environments in Java language, by providing an interface between agents and the programmed environment. These environments, however, are centralised, and so they are meant for small systems or specific test scenarios. This hinders scalability, which is an important aspect to consider when working with complex, more realistic scenarios or simply more robust structures. To address this limitation, we use the CArtAgO framework for environment programming.

In CArtAgO, environments are not seen as a centralised domain description, but as a distribution of observable properties and operations among artefacts. These artefacts represent objects in the environment through which agents interact with one another indirectly, *e.g.* a table in an office, on which an agent may stack reports for another agent to pick these reports up and read them. The artefact model is useful because it groups operations according to a context, so it is not only easier to understand the environment model, but also to maintain it. Agents can create and destroy artefacts at their convenience, and should new operations be needed for a new feature in the MAS, it can be done by adding new artefacts, instead of changing existing routines to conform to new protocols. This approach is also more scalable, as one of the basic features of CArtAgO is that it can distribute artefacts among workspaces. Workspaces are artefact containers that can be configured in several nodes in a network, eliminating the need to concentrate the environment on a single machine. In our work, we use artefacts for offering monitoring and enforcement tasks to agents, and we refer to these artefacts as "normative artefacts". These normative artefacts are shared between normative agents so that more monitors and enforcers may be added to the system as it scales up.

4.3 Norms

In order to keep competition between agents manageable a designer creates norms to direct agent behaviour and maintain environment stability. This is achieved by specifying obligations and prohibitions [6]. Here, obligations are behaviours that agents must follow in a given context to comply with the norm, and prohibitions behaviours that jeopardise the environment's stability, and so must be avoided. Violating prohibitions is just as harmful as violating obligations, hence both cases must be addressed when detected. We expect that, when agents are punished for transgression, they are able to learn not to misbehave. Examples 1 and 2, in Sect. 3, correspond to an obligation and a prohibition, respectively.

While norms in the real world are expressed in natural language, they must be translated to a multi-agent environment so that agents are able to reason about them. This requires the extraction of necessary information related to a norm and composition of a mathematical representation. Agents should not have to reason how or why a certain norm came to be, but rather what the norm is about and what are the consequences of violating it. The format can also be extended to include other important information, such as the sanction function associated with a norm's violation, or the conditions for automatic activation and expiration of the norm [6]. In this paper, norms as specified according to the tuple of Definition 1.

Definition 1. *A norm is represented by the tuple $\mathcal{N} = \langle \mu, \kappa, \chi, \tau, \rho \rangle$, where:*

- $\mu \in \{obligation, prohibition\}$ *represents the norm's modality.*
- $\kappa \in \{action, state\}$ *represents the type of trigger condition enclosed.*
- χ *represents the set of states (context) to which a norm applies.*
- τ *represents the norm's trigger condition.*
- ρ *represents the sanction to be applied to violating agents.*

Using Definition 1, we can proceed to formalising the norms from our example. We can formalize the first norm of our scenario from Example 1, as shown in Example 3.

Example 3. $\langle obligation, action, valid(Passport), accept(Passport), loss(5) \rangle$

The process can be repeated for Example 2. By identifying the context of a norm, it is possible to define it solely with predicates and atoms, as shown in Example 4, below.

Example 4. $\langle prohibition, action, not\ valid(Passport), accept(Passport), loss(10) \rangle$

4.4 Action Records

Like norms, actions must also be stored as tuples containing essential information. Actions captured by monitors must only be accessed by agents of the enforcer type, and therefore only the pieces of information that can be associated with norms are deemed essential. These are: what was done; who did it; and under what context it was done. Example 5 shows how a monitor reports its observations to an enforcer:

Example 5. "Officer John Doe approved Passport #3225. The passport was known to be valid."

From this report, we can extract the following details:

Example 6. $\langle johndoe, approve(Passport), valid(Passport)\rangle$

In this example, an officer approves the entry of an immigrant holding a valid passport. The next report reads:

Example 7. "Officer John Smith approved Passport #2134. The passport's validity could not be confirmed."

From this report, we can extract the following details:

Example 8. $\langle johnsmith, approve(Passport), notvalid(Passport)\rangle$

As such, we define Action Records:

Definition 2. *An Action Record, stored within the Action History, is represented by the tuple: $\mathcal{R} = \langle \gamma, \alpha, \beta\rangle$, where:*

- *γ represents the agent executing the action;*
- *α is the action description in the form "$f(p_0, p_1, \ldots, p_n)$", where f is an action name and $p_0, \ldots, p_n$ are the action's parameter values; and*
- *β represents agent γ's beliefs at the moment of execution.*

4.5 Monitoring System

The monitoring is divided in two parts: a capturing system, which gathers information pertaining to an action's execution context, and a report forwarding system, which provides enforcers with the gathered information for violation detection. To gather relevant information, the capturing system employs two strategies: an action capturing strategy and a belief state capturing strategy. In action capturing, whenever an agent successfully executes an action, the capturing system takes note of that action. In CArtAgO, this means that each successful operation is recorded for further analysis. Should an action fail for any reason, the capturing system ignores it. Yet, recording every successful action is a problem for both scalability and practicality. There is no reason to capture actions that are not enforced by any norm, *e.g* book-keeping actions or CArtAgO's own artefact creation and lookup operations. As such, we include capturing routines only for the operations relevant to the normative context, so as not to waste neither space and time with unimportant actions.

In belief state capturing, we employ a similar strategy to that of action capturing. Much like actions, there may be beliefs which are not related to any norms in the system. Thus, we should apply a filtering procedure when scanning beliefs to avoid wasting space on useless information. We propose a simple filtering technique, which requires monitors to also focus on normative activity:

1. For each new active norm, scan the norm's context for literals to add to a *to be observed* list.
2. If the norm's triggering condition is of the state type, do the same with the condition's literals.
3. For each deactivated norm, remove it from the *to be observed* list only literals that are not seen in any other norms.

We then change our capturing routines to scan the belief bases only for the literals in the *to be observed* list. If any belief *to be observed* cannot be found in the belief base, they can be ignored. Note that this list can contain only predicate names, and not their full list of terms.

Once we capture an action, we store it in the **Action History**, which is a queue-like data structure from which monitors gather information to build the reports that they send to enforcers. Actions are stored in the format discussed in Sect. 4.4 and are removed from the queue as soon as a monitor attempts to read them, regardless of the monitor's success in doing so.

It is the monitors's responsibility to send captured actions to enforcer agents in the form of a report for analysis. To achieve that, we use a producer/consumer model, in which an agent continuously provides information, through a channel, to another agent that consumes this information. With this in mind, we can identify four components that are necessary for this setup: a Producer, a Consumer, a channel for communications and the information itself. In our context, the role of Producer is given to Monitor Agents; the role of Consumer is given to Enforcer Agents; the communication channels are artefacts called "Reporting Interface"; and the information that transits through this channel are reports containing the actions executed by agents. This process is illustrated in Fig. 2.

Since monitoring in the real world is not cost-free, we need to spend resources to have an effective monitoring system in place [26], with the effectiveness of a monitor depending on its intensity. For this reason, we must enable the adjustment of monitoring intensity, so that enforcement can be performed at a cost considered affordable by the society. These adjustments take the form of different monitoring strategies. An example would be a probabilistic strategy, in which each captured action has a probability of being successfully read by a monitor. If the reading is successful, the action is guaranteed to be reported to an enforce, whereas if the reading fails, then the action is lost forever. We can use this to simulate the imperfect monitoring of actions, when some violations may go unpunished. Other strategies that monitors may apply include reading

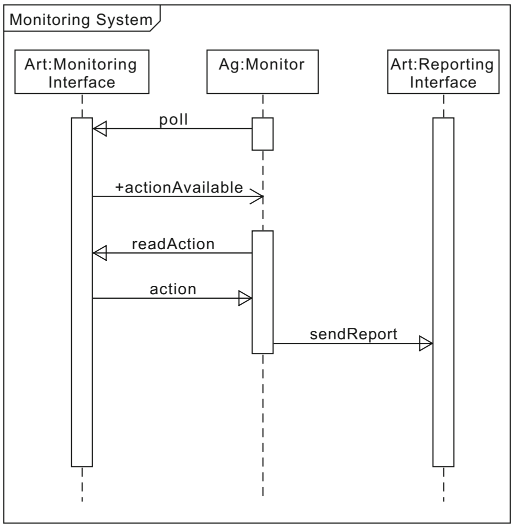

Fig. 2. Monitors poll the monitoring interface for new actions. When a monitor is successful at reading an action, it sends a report containing the action for analysis via the reporting interface.

only actions that they know are being enforced by an active norm. An extension to this strategy would be to add a probability of reading enforced actions with success. In this paper, we use the probabilistic strategy to study the general behaviour of our simulation.

4.6 Enforcement System

The enforcement system represents the Consumer entity in the normative mechanism's Producer/Consumer scheme. An enforcer agent connects to the Reporting Interface and awaits the arrival of new reports to analyse. The arrival of new reports is perceived by the enforcer, and in our implementation this perception is mapped to the `+newReport` signal. Once the report submission is perceived, the enforcer accesses the Normative Interface in search of currently activated norms and checks for any possible violations by the reported action.

During the violation detection routine, the perception of violations is also mapped to a signal, represented in the sequence diagram of Fig. 3 as the `+violation` event. When a violation is perceived, it falls to the enforcer to apply associated sanctions. The sanctioning step is the last in this process, and it starts as soon as detection finishes.

In order to sanction violating agents, the normative mechanism must be able to recognise them. It does not make sense to be told "John has approved an invalid passport. He violated a norm". if we do not know who John is in the first place. Therefore agents must be registered to the normative system prior to execution of their designed plans, similar to how people are registered for

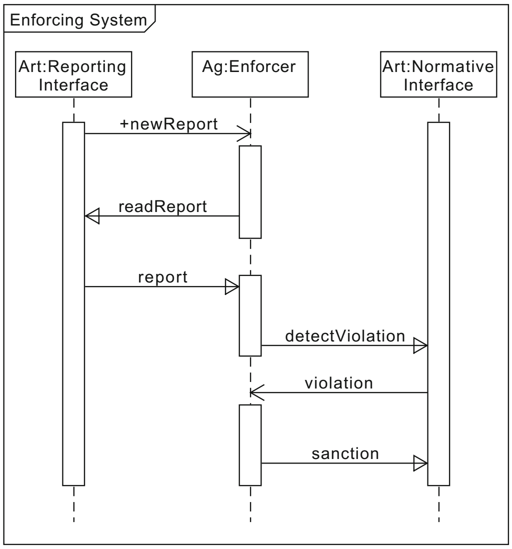

Fig. 3. Enforcers read new reports via the reporting interface. For each report, they use the normative interface to access the normative base and look for violations in agents's actions.

government issued IDs. In CArtAgO, this is accomplished through an operation in the Normative Interface that adds the agent's ID to a list, so that they may be found when needed. The ID they are registered with should be the same that appears in Action Records.

Normative Base. When norms are created, they must be stored within the system so that they may be accessed by an enforcer attempting to detect violations. The Normative Base structure holds all the norms that exist in the system, active or not. Every time a norm is created, it is stored in a list structure with a unique identifier. Norms may be activated or deactivated through the Normative Interface. Every time a norm is created, activated, deactivated or destroyed, agents connected to the Normative Interface perceive the event.

Detecting Violations. The detection operation runs for each action report received by an enforcer agent. Each action read is verified against the normative base, along with the context under which the action was executed. Since it is possible for an action to violate more than one norm, we utilize a list structure to take note of all violations detected so they will be properly addressed at a later time. At first, no norm is seen as violated and thus the list is empty. A norm is only added to the list when all verification steps finish with the variable's $isViolated$ value set to $True$. The procedure for detecting violations can be seen in Algorithm 1 and is explained further.

Algorithm 1. Violation detection algorithm.

```
1: function DetectViolation($\langle\gamma, \alpha, \beta\rangle$)
2:     $V \leftarrow [\,]$
3:     for each $n = \langle\mu, \kappa, \chi, \tau, \rho\rangle \in ActiveNorms$ do
4:         if ContextApplies($\chi, \beta$) then
5:             if ConditionApplies($\kappa, \tau, \alpha, \beta$) then
6:                 if $\mu = prohibition$ then
7:                     $V \leftarrow V \cup \{n\}$    ▷ Violation detected! Adds to the list of violated norms.
8:             else
9:                 if $\mu = obligation$ then
10:                    $V \leftarrow V \cup \{n\}$    ▷ Violation detected! Adds to the list of violated norms.
11:    for each $n \in V$ do
12:        SignalViolation($n, \gamma$)
```

Detection of violations can be achieved in two steps: context analysis and trigger condition analysis. Context analysis is about making sure that the action's execution context is the same as the one predicted by a norm. If it is, then there is a possibility of violation and further analysis is required. Otherwise, violation is considered an impossibility and the routine carries on. Formally, we define the norm's context as χ and the acting agent's belief-base as β. Hence, the context analysis returns $True$ value if $\chi \subseteq \beta$. Algorithm 2 is used for comparing sets of predicates. It checks if all the predicates defined in context χ are present in the agent's belief-base β, one by one. If a predicate in χ is negated (*e.g* `not valid(Passport)`), then the algorithm checks for its absence in belief-base β instead. This is to reflect how the `not` operator works in Jason. The routine returns $True$ if the trigger condition is satisfied and $False$ otherwise.

A trigger condition of a norm can be either the execution of an action or the achievement of a state by an agent. This is specified by the norm's trigger condition type and directs the way in which the detection algorithm executes. If we are working with an action trigger, then we must compare the action that was executed with the one specified by the norm. However, if we are working with a state trigger, then two contexts must be compared: the agent's belief-base and the norm's state trigger condition. These are compared using the context analysis algorithm of Algorithm 2. We show the pseudo-code for the trigger analysis procedure in Algorithm 3.

When both context and trigger conditions are satisfied, we need only verify whether the norm is an obligation or prohibition to conclude if it was violated or not. A prohibition means that a certain action or state is undesired under the given context. If all the conditions up to now have been met, we conclude that said undesired state has been reached and the norm was violated. On the other hand, an obligation requires the flow specified by the norm to be followed strictly, and if this is the case, we conclude that the norm was complied with. By negating our conditions, we also negate its results: if in a prohibition context the conditions were not met, then we would be home free; if they are not met while in an obligation context, however, we would have just violated it.

Algorithm 2. Context comparison sub-routine.

```
1: function ContextApplies($\chi = [l_1, ..., l_n], \beta = [l_1, ..., l_n]$)
2:     Require $count(\chi) \leq count(\beta)$
3:     for each $p \in \chi$ do
4:         $isPresent \leftarrow False$
5:         $checkAbsence \leftarrow False$
6:         if $p$ is of the form $\neg\phi$ then
7:             $p \leftarrow \phi$
8:             $checkAbsence \leftarrow True$
9:         for each $l \in \beta$ do
10:            if $l = p$ then
11:                $isPresent \leftarrow True$
12:                break
13:        if $checkAbsence = isPresent$ then
14:            return $False$
15:    return $True$
```

Their modality notwithstanding, every norm that is violated is added to a list that is processed when all norms have been verified. Sanction functions are then executed and agents perceive their punishments. Penalties can be brought directly upon agents through perception or carried out by a third party, while records on agent transgressions can be maintained in a separate structure for greater consistency.

Algorithm 3. Trigger condition analysis sub-routine.

```
1: function ConditionApplies($\kappa, \tau, \alpha, \beta$)
2:     if $\kappa = action$ then
3:         return $\tau = \alpha$
4:     return ContextApplies($\tau, \beta$)
```

5 Evaluation

In order to test our solution, we developed agents using Jason and deployed them in a CArtAgO environment following the scenario described in Sect. 3. To visualise the difference between compliant and non-compliant behaviours, two types of agents were used: the *normal* type and the *corrupt* type. The normal type is programmed to approve only those passports that are truly valid, whereas the corrupt one will approve passports indiscriminately for his own personal gain. By making it so, we can more easily tell the effectiveness of the norm enforcing mechanism. Therefore, the following results were expected:

– Corrupt agents attain more credits when under lower monitoring intensity.
– Standard agents maintain an average quantity of credits through all simulations.
– At some point, corrupt agents should start performing poorly due to higher monitoring intensity. This marks the point at which monitoring can change the environment.

We ran 35 experiments for 11 different values of monitoring intensity[2]. Intensity values range from 0 to 100, with a step value of 10. Each simulation was run for 10 min. In this timespan, with our set-up, around 1048 immigrants attempt to cross the border. In what follows, we refer to an agent's obtained credits, or their performance measure, as their utility. We use that measure in the graph of Fig. 4, which illustrates how the environment's monitoring intensity affects the utilities of corrupt agents 1 and 2. The monitoring intensity is the probability as a percentage of a monitor being able to read an agent's action. A value of 100 means that all actions are read, while a value of 0 means no actions are read by the monitor. We notice that, as the intensity of the monitoring mechanism increases, the utility of corrupt agents decreases to the point where performing badly and not performing at all yield the same utility, whereas normal agents maintain their average utility. This allows us to conclude that, for a monitoring intensity value of 40 or more, following norms is a better decision than the contrary.

The data used to plot the graph of Fig. 4 can be seen in Table 1. Values for μ and σ represent the arithmetic mean and standard deviation, respectively. These were calculated to show that utility values for normal agents are near constant. The μ values for corrupt agents show that, at the end of the simulation, their average performance is worse than those of normal agents, due to their constant violation of norms. A high σ value for these agents shows that their performance suffers between simulations. We can then see that through the analysis of recorded agent actions and successful identification of violation occurrences, violating agents are punished by the enforcement system and have their utilities affected.

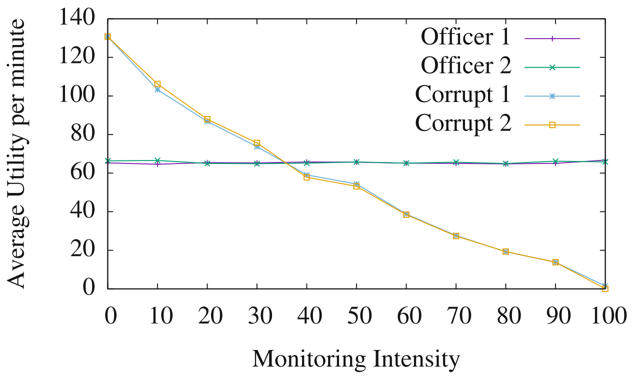

Fig. 4. Utility of corrupt agents is affected by monitoring intensity.

[2] Although our experiments correspond in a broad sense to a simulation, we avoid the term for its possibly loaded meaning.

Table 1. Agent utilities × monitoring intensity.

Intensity	Officer1	Officer2	Corrupt officer1	Corrupt officer2
0	65,3285	66,3714	130,6571	130,7000
10	64,5871	66,5714	103,3000	106,2285
20	65,4428	65,0142	86,8000	87,9571
30	65,3142	64,8714	73,7571	75,6571
40	65,7857	65,1857	59,0571	57,8142
50	65,6714	65,7714	54,3285	53,1857
60	65,1571	65,1714	38,7714	38,4571
70	65,0142	65,6571	27,6428	27,3714
80	64,7857	64,9571	19,2285	19,3428
90	65,0714	66,1714	13,7857	13,8142
100	66,7571	65,8000	1,4714	0,0285
μ	65.3559	65.5948	55,3454	55,5051
σ	0.5569	0.5705	38,4836	39,1996

6 Related Work

There are multiple tools available for programming multi-agent environments, few of which provide mechanisms for norm specification. These tools range from programming libraries to model-based simulators. To name a few, NetLogo [16] and its distributed version HubNet [27] are of the model-based type and allow users to work with educational projects and, to some extent, professional ones. Other tools include MASON [17] and Repast [18]. MASON is a simulation library developed in Java that provides functions for modelling agents and visualising simulations as they run. As for Repast, it uses interaction models much like NetLogo does, although it is meant for professional use and thus offers more alternatives for agent programming. One final example worth mentioning is MASSim [11], which promotes multi-agent research and is used in the MAS Programming Contest[3] [28]. This one, however, provides only the tools related to the contests. Although it is possible to develop custom agents for operation within the simulator, the practice is not encouraged by its developers.

Building a full-fledged norm-based behaviour simulation engine is not a trivial task, and the "Emergence in the Loop" (EMIL) [29] project built a set of tools to accomplish this objective. A toolset which includes an extension of the BDI architecture that is capable of simulating the processes referred to as "immergence" and "emergence" of norms [30]; and an integration with multi-agent modelling tools such as NetLogo [16] and Repast [18]. In this way, agents are modelled in one of these environments and then simulated using the EMIL agent architecture. It is a very powerful tool for studying social behaviour in

[3] https://multiagentcontest.org.

autonomous agents, since agents can reason about norms and, together, create conventions of what kinds of behaviours must be avoided or followed. EMIL's approach to normative simulation is more focused on agents and their experience with norms. This contrasts with our approach in that we are more focused on norm monitoring and enforcement tasks, and little is said about these matters in the EMIL literature. We also consider the environmental aspects of Normative Multi-Agent Systems, which is why we employ CArtAgO in our implementation.

Finally, the $\mathcal{M}$oise$^+$ [19] tool (part of the JaCaMo [31] framework) can also be used to specify norms for MAS development. $\mathcal{M}$oise$^+$ allows us to create organisations of agents, and within these organisations agents take up specific roles to act and missions to accomplish. The normative part of $\mathcal{M}$oise$^+$ ties agents to their missions through obligations, prohibitions and permissions. Nevertheless, $\mathcal{M}$oise$^+$ differs from NormMAS in three key aspects. First $\mathcal{M}$oise$^+$ focuses on normative specification for organisations to coordinate agents in performing certain tasks, whereas in NormMAS, we have social norms and regulations that only tell agents what they should or should not do. Consequently, when there is no normative specification in NormMAS, the agents's routines remain intact. Second, while $\mathcal{M}$oise$^+$ norms affect whole plans, NormMAS norms affect only specific actions or states. Third, while $\mathcal{M}$oise$^+$ norms are not regimented, lack of compliance does not incur any penalties for violating agents, which means that they are not enforced either.

7 Conclusions and Future Work

In this paper, we constructed a mechanism of norm processing and enforcement in a multi-agent environment. We show its feasibility with an implementation using Jason [12] and Cartago [22] technologies. By keeping track of agent activities and analysing actions against a normative base, it is possible to detect violations and enforce norms through the sanctioning of violating agents. With this framework, it is possible to evaluate different implementations [6,32–34] of normative behaviour. Statistics collection can also be customised so that results may be compared between simulations. We provide our example implementation to the public via a GitHub repository [35].

CArtAgO allows us to build environments in a distributed manner, therefore providing scalability for realistic simulation scenarios or complex multi-agent systems. The philosophy behind CArtAgO, which sees the environment as the composition of artefacts through which agents interact, also aided in the framework's construction. Artefacts are modular, they can be attached or detached to a multi-agent system seamlessly. Meaning that artefacts can be created to suit an agent's or group of agents's specific needs, and agents may connect only to those artefacts that are related to their designs. We took advantage of those features to build the interfaces for the monitoring system to access the Action History and Normative Base structures.

As future work, we aim to build improvements and extensions to the framework, such as: a mechanism to be added to the normative system that allows activation and expiration of norms following predefined conditions; agent architectures that can learn from normative environments, and with that avoid penalties by violation or minimising performance loss when violations are inevitable [6]; enable agents to learn about the enforcing intensity and use that information to their advantage [26]; and the introduction of agent hierarchies to control normative power [36].

References

1. Wooldridge, M.: Intelligent agents. In: Weiss, G. (ed.) Multi-Agent Systems, 2nd edn, pp. 3–50. The MIT Press, Cambridge (2013)
2. Jones, A.J.I., Sergot, M.: On the characterisation of law and computer systems: the normative systems perspective. In: Meyer, J.-J.C., Wieringa, R.J. (eds.) Deontic Logic in Computer Science: Normative System Specification, Wiley Professional Computing Series, Chapter 12, pp. 275–307. Wiley, Chichester (1993)
3. Kollingbaum, M.: Norm-governed practical reasoning agents. Ph.D. thesis, University of Aberdeen (2005)
4. Broersen, J., Dastani, M., Hulstijn, J., Huang, Z., van der Torre, L.: The BOID architecture: conflicts between beliefs, obligations, intentions and desires. In: Proceedings of the Fifth International Conference on Autonomous Agents, pp. 9–16 (2001)
5. Governatori, G., Rotolo, A.: BIO logical agents: norms, beliefs, intentions in defeasible logic. Auton. Agent. Multi-Agent Syst. **17**(1), 36–69 (2008)
6. Meneguzzi, F., Luck, M.: Norm-based behaviour modification in BDI agents. In: Proceedings of the Eighth International Conference on Autonomous Agents and Multiagent Systems, pp. 177–184 (2009)
7. Criado, N.: Using norms to control open multi-agent systems. Ph.D. thesis, Universitat Politécnica de València (2012)
8. Alechina, N., Dastani, M., Logan, B.: Programming norm-aware agents. In: van der Hoek, W., Padgham, L., Conitzer, V., Winikoff, M., (eds.) Autonomous Agents and Multi-Agent Systems, IFAAMAS, pp. 1057–1064 (2012)
9. Panagiotidi, S., Vázquez-Salceda, J., Dignum, F.: Reasoning over norm compliance via planning. In: Aldewereld, H., Sichman, J.S. (eds.) COIN 2012. LNCS, vol. 7756, pp. 35–52. Springer, Heidelberg (2013)
10. Meneguzzi, F., Mehrotra, S., Tittle, J., Oh, J., Chakraborty, N., Sycara, K., Lewis, M.: A cognitive architecture for emergency response. In: Proceedings of the Eleventh International Conference on Autonomous Agents and Multiagent Systems, pp. 1161–1162 (2012)
11. Behrens, T.M., Dastani, M., Dix, J., Novák, P.: MASSi: multi-agent systems simulation platform. In: Begehung des Simulationswissenschaftlichen Zentrums. Clausthal University of Technology (2008)
12. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak Using Jason (Wiley Series in Agent Technology). Wiley, Chichester (2007)
13. Bellifemine, F.L., Caire, G., Greenwood, D.: Developing Multi-Agent Systems with JADE (Wiley Series in Agent Technology). Wiley, New York (2007)
14. Conte, R., Andrighetto, G., Campennl, M.: Minding Norms: Mechanisms and Dynamics of Social Order in Agent Societies. Oxford Series on Cognitive Models and Architectures. OUP, Oxford (2013)

15. Fagundes, M., Ossowski, S., Meneguzzi, F.: Analyzing the tradeoff between efficiency and cost of norm enforcement in stochastic environments populated with self-interested agents. In: Proceedings of the 21st European Conference on Artificial Intelligence (2014)
16. Wilensky, U.: NetLogo. Center for Connected Learning and Computer-Based Modeling, Northwestern University. Evanston, IL (1999). http://ccl.northwestern.edu/netlogo/
17. Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K., Balan, G.: MASON: A multi-agent simulation environment. Simulation **81**, 517–527 (2005)
18. North, M., Collier, N., Ozik, J., Tatara, E., Macal, C., Bragen, M., Sydelko, P.: Complex adaptive systems modeling with repast simphony. Complex Adapt. Syst. Model. **1**(1), 1–26 (2013)
19. Hubner, J.F., Sichman, J.S., Boissier, O.: Developing organised multiagent systems using the MOISE+ model: programming issues at the system and agent levels. Int. J. Agent-Oriented Softw. Eng. **1**, 370–395 (2007)
20. Jones, A.J.I., Sergot, M.: On the characterisation of law, computer systems: the normative systems perspective. In: Deontic Logic in Computer Science: Normative System Specification, pp. 275–307. Wiley (1993)
21. Oren, N., Vasconcelos, W., Meneguzzi, F., Luck, M.: Acting on Norm Constrained Plans. In: Leite, J., Torroni, P., Ågotnes, T., Boella, G., van der Torre, L. (eds.) CLIMA XII 2011. LNCS, vol. 6814, pp. 347–363. Springer, Heidelberg (2011)
22. Ricci, A., Viroli, M., Omicini, A.: CArtAgO: a framework for prototyping artifact-based environments in MAS. In: Weyns, D., Dyke Parunak, H., Michel, F. (eds.) E4MAS 2006. LNCS (LNAI), vol. 4389, pp. 67–86. Springer, Heidelberg (2007)
23. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: Perram, J., Van de Velde, W. (eds.) MAAMAW 1996. LNCS, vol. 1038, pp. 42–55. Springer, Heidelberg (1996)
24. Bratman, M.E.: Intention, Plans and Practical Reason. Harvard University Press, Cambridge (1987)
25. Meneguzzi, F., De Silva, L.: Planning in BDI agents: a survey of the integration of planning algorithms and agent reasoning. Knowl. Eng. Rev. **30**, 1–44 (2015)
26. Meneguzzi, F., Logan, B., Fagundes, M.S.: Norm monitoring with asymmetric information. In: Bazzan, A.L.C., Huhns, M.N., Lomuscio, A., Scerri, P. (eds.) International Conference on Autonomous Agents and Multi-Agent Systems, AAMAS 2014, Paris, France, pp. 1523–1524. IFAAMAS/ACM, 5–9 May 2014
27. Wilensky, U., Stroup, W.: HubNet. Center for Connected Learning and Computer-Based Modeling, Northwestern University. Evanston, IL (1999). http://ccl.northwestern.edu/netlogo/hubnet.html
28. Behrens, T.M., Dastani, M., Dix, J., Hübner, J., Köster, M., Novák, P., Schlesinger, F.: The multi-agent programming contest. AI Mag. **33**(4), 111–113 (2012)
29. Andrighetto, G., Conte, R., Turrini, P., Paolucci, M.: Emergence in the loop: simulating the two way dynamics of norm innovation. In: Boella, G., van der Torre, L.W.N., Verhagen, H. (eds.) Normative Multi-agent Systems, vol. 07122, Dagstuhl Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum für Informatik, Schloss Dagstuhl, Germany, 18–23 March 2007
30. Andrighetto, G., Campennì, M., Conte, R., Paolucci, M.: On the immergence of norms: a normative agent architecture. In: Proceedings of the Association for the Advancement of Artificial Intelligence Symposium, Social and Organizational Aspects of Intelligence, Forthcoming (2007)

31. Boissier, O., Bordini, R.H., Hubner, J.F., Ricci, A., Santi, A.: Multi-agent oriented programming with JaCaMo. Sci. Comput. Prog. **78**(6), 747–761 (2013). Special section: The Programming Languages track at the 26th ACM Symposium on Applied Computing (SAC 2011); Special section on Agent-oriented Design Methods and Programming Techniques for Distributed Computing in Dynamic and Complex Environments
32. Lee, J., Padget, J., Logan, B., Dybalova, D., Alechina, N.: Run-time norm compliance in BDI agents. In: International Conference on Autonomous Agents and Multi-Agent Systems, AAMAS 2014, pp. 1581–1582 (2014)
33. Vasconcelos, W.W., Kollingbaum, M.J., Norman, T.J.: Normative conflict resolution in multi-agent systems. Auton. Agents Multi-Agent Syst. **19**(2), 124–152 (2009)
34. Criado, N., Argente, E., Botti, V.J., Noriega, P.: Reasoning about norm compliance. In: Sonenberg, L., Stone, P., Tumer, K., Yolum, P. (eds.) 10th International Conference on Autonomous Agents and Multiagent Systems, Taipei, Taiwan, vol. 1–3, pp. 1191–1192, IFAAMAS, 2–6 May 2011
35. Chang, S.: normmas-sim: NormMAS - paper version. Zenodo, December 2015. doi:10.5281/zenodo.35028
36. Oren, N., Luck, M., Miles, S.: A model of normative power. In: van der Hoek, W., Kaminka, G.A., Lespérance, Y., Luck, M., Sen, S. (eds.) 9th International Conference on Autonomous Agents and Multiagent Systems, Toronto, Canada, vol. 1–3, pp. 815–822, IFAAMAS, 10–14 May 2010