A Framework for Programming Norm-Aware Multi-Agent Systems

Daniela Dybalova[1], Bas Testerink[2], Mehdi Dastani[2], and Brian Logan[1]

[1] School of Computer Science, University of Nottingham,
Nottingham, UK
{dxd,bsl}@cs.nott.ac.uk
[2] Department of Information and Computing Sciences, Universiteit Utrecht,
Utrecht, The Netherlands
{b.j.g.testerink,M.M.Dastani}@uu.nl

Abstract. We propose a programming framework for the implementation of *norm-aware multi-agent systems*. The framework integrates the N-2APL norm-aware agent programming language with the 2OPL organisation programming language. Integration of N-2APL and 2OPL is achieved using a tuple space which represents both the (brute) state of the multi-agent environment and the detached norms and sanctions comprising its normative state. To the best of our knowledge, this is the first implementation of an integrated framework for norm-aware MAS in which autonomous agents deliberate about whether to conform to the norms imposed by a normative organisation. The use of a tuple space makes it straightforward to integrate other system components. To illustrate the flexibility of our framework, we briefly describe its application in a novel normative application, a mixed reality game called GeoSense. We show how GeoSense game rules can be expressed as conditional norms with deadlines and sanctions, and how agents can deliberate about their individual goals and the norms imposed by the game.

1 Introduction

Norms can be viewed as defining standards of behaviour. They have been widely proposed as a means of coordinating and regulating the behaviours of individual agents to ensure global properties of a multi-agent system. For example, smart roads may be implemented as multi-agent systems, where autonomous cars are agents and the road infrastructure constitutes the agents' environment. Desirable properties of such a multi-agent system may include safety, road throughput, and minimal environmental damage. Such properties can be ensured by means of enforcement and regimentation of traffic norms such as speed limits, redirecting traffic, and closing road lanes. Multi-agent systems that use norms to regulate agent behaviour are called normative multi-agent systems.

In building normative multi-agent systems, norms can be implemented either endogenously by integrating them into the programs of individual agents (e.g., an autonomous car may be programmed not to exceed the speed limit) or exogenously by additional components that observe and evaluate the agents' behaviours in order to check compliance or violation of norms (e.g., road cameras monitor cars' speed and

register the identities of cars that violate speed limitations). In exogenous normative multi-agent systems, norms can regulate the behaviour of agents by means of regimentation or enforcement. Norm regimentation prevents agents from violating norms (e.g., closing lanes of a smart road) while norm enforcement allows agents to violate norms but imposes sanctions on violating agents to compensate for their violations (e.g., violating the speed limit incurs a sanction in the form of a fine). In multi-agent systems where norms are implemented exogenously, regulation is realized by processing norms at run time. The processing of norms in such systems requires creating and eliminating norms based on their conditions and deadlines, monitoring the activities of participating agents, evaluating their behaviour with respect to the specified norms and finally determining appropriate consequences for the participating agents. In multi-agent systems where norms are implemented endogenously, individual agents have internalized norms in the sense that their decision procedures are defined in terms of the norms. Although the agents' decisions in such systems do not necessarily need to be norm compliant, it is not clear how to cope with norm violations without an external entity that detects norm violations and compensates them by means of sanctions. It is also important to emphasize that not every norm can be regimented exogenously. In the smart road example, it is not clear how speed limits can be regimented in the highways since placing speed bumps is not a realistic option.

A number of programming frameworks have been proposed for the development of normative multi-agent systems, e.g., [1, 2]. However in these frameworks, the agents do not deliberate about whether to comply with norms. In [3] an agent programming language N-2APL, for programming *norm-aware* agents was introduced. Norm-aware N-2APL agents are able to deliberate on their goals, norms and sanctions before deciding which plan to select and execute, and are able to violate norms if it is in their overall interest to do so, e.g., if meeting an obligation would result in an important goal of the agent becoming unachievable.[3]

In this paper we propose a framework for programming norm-aware multi-agent systems. The framework integrates the N-2APL agent programming language with the 2OPL language for programming normative organisations. The integration of N-2APL and 2OPL is achieved using a tuple space which represents both the (brute) state of the multi-agent environment and the detached norms and sanctions comprising its normative state. To the best of our knowledge, this is the first implementation of N-2APL and the first implementation of an integrated framework for norm-aware multi-agent systems in which autonomous agents deliberate about whether to conform to the norms imposed by a normative organisation. The use of a tuple space makes it straightforward to integrate other system components. To illustrate the flexibility of our framework, we briefly describe its application in a novel normative application, a mixed reality game called GeoSense [6]. We show how GeoSense game rules can be expressed as conditional norms with deadlines and sanctions, and how agents can deliberate about their individual goals and the norms imposed by the game.

[3] Norm-aware agents are related to the notion of *deliberate normative agents* in [4], and are capable of *behaving according to a role specification in a normative organization* and *reasoning about violations* in the sense of [5].

The remainder of this paper is structured as follows. In Section 2 we introduce our programming framework. We briefly describe 2OPL and N-2APL and their implementations, and explain how they are integrated using the JavaSpaces tuple space. In Section 3 we briefly describe the application of our framework to allow norm-aware agents to play the mixed reality game GeoSense. We briefly outline the translation of game rules into 2OPL norms, how N-2APL agent programs encode the game play of the agents, and the integration of the resulting normative multi-agent system with the GeoSense game server. We discuss related work in Section 4 and conclude in Section 5.

2 Framework Description

In this section we describe our framework, which consists of three main parts: a 2OPL normative organization, a N-2APL multi-agent system and a Linda-like tuple space which acts as a coordination mechanism.

2.1 2OPL Normative Organization

2OPL [1, 7] is a programming language designed to support the implementation of normative multi-agent systems where norms are implemented exogenously. 2OPL programs contain three types of data: facts, fact update rules, and norms. The facts and fact update rules are used to represent the state of the agents' environment and the effect of the agents' actions in the environment. For example, in the GeoSense game, a fact may represent the current location of an agent, while a fact update rule represents how the agent's location changes as a result of performing a 'move' action.

2OPL norms are state-based norms and are defined in terms of a unique label, an activation condition, and a deontic element.[4] The label functions as a name that can be used to refer to the norm and the precondition specifies when (i.e., in which states of the environment) the norm can be activated (detached). The deontic element of the norm is either an obligation or a prohibition. An obligation is defined by a subject (the agent to which incurs the obligation), a deadline, a state formula indicating the state of the environment that has to be brought about before the deadline, and a sanction formula indicating how the state is updated if the obligation is not discharged by the deadline. A prohibition is defined by a state formula indicating the state of the environment that must be avoided, and a sanction formula indicating how the state is updated if the prohibition is violated before the deadline. The subject and deadline are represented by atoms, and the state and sanction formulas are represented as conjunctions (lists) of atomic facts. For example, in the GeoSense game, a norm may prohibit the truck from entering a specific area, with violation of the norm resulting in the truck's score being reduced by 500 points.

The syntax of 2OPL norms is shown in Figure 1 where the $\langle atom \rangle$ follows the Prolog syntax for atomic facts. All components of the norm must be ground when a norm instance is detached. For integration with N-2APL agents, we require that 2OPL

[4] In what follows, we adopt the version of 2OPL described by Tinnemeier [7], which includes conditional norms with deadlines.

$\langle Norm \rangle$ ::= “norm(” $\langle label \rangle, \langle precond \rangle, \langle deontic \rangle$ “)”
$\langle label \rangle$ ::= $\langle atom \rangle$
$\langle precond \rangle$ ::= “(” $\langle atom \rangle$ (“,” $\langle atom \rangle$)* “)”
$\langle deontic \rangle$ ::= “obligation(” $\langle subject \rangle, \langle state \rangle, \langle deadline \rangle, \langle sanction \rangle$ “)” |
“prohibition(” $\langle subject \rangle, \langle state \rangle, \langle deadline \rangle, \langle sanction \rangle$ “)”
$\langle subject \rangle$ ::= $\langle atom \rangle$
$\langle state \rangle$ ::= “[” $\langle atom \rangle$ (“,” $\langle atom \rangle$)* “]”
$\langle deadline \rangle$ ::= $\langle atom \rangle$
$\langle sanction \rangle$ ::= “[” $\langle atom \rangle$ (“,” $\langle atom \rangle$)* “]”

Fig. 1. Syntax of 2OPL norms

norms conform to a more restrictive syntax than that shown in Figure 1. In particular, we assume a global clock and require that deadlines are atoms denoting relative times after the time at which a norm is detached. We also require that prohibitions have a deadline of infinity. These restrictions are necessary to ensure that the normative reasoning of N-2APL agents remains tractable [3]. In addition, to simplify the mapping from sanctions to the priorities N-2APL agents assign to goals (see below), we assume that sanctions are single atoms.

2OPL programs are executed by means of an interpreter that consists of a loop in which agents' actions are observed, the effects of the actions are realized by means of the fact update rules, and norms are processed. Norms are processed as follows. If the precondition of a norm holds, then an instance of the corresponding obligation or prohibition is detached (comes into effect). For all obligations that are already in effect, the 2OPL interpreter checks if the deadline is reached while the obliged state of the environment is not realized. In such a case a violation has occurred, and the state of the environment is updated with the corresponding sanction. Moreover, for all prohibitions that are already in effect it is checked if the prohibited state is realized. In such a case a violation has occurred and the state of the environment is updated with the corresponding sanction.

To support the integration of 2OPL with the framework, the 2OPL interpreter was extended to interact via a tuple space as described in Section 2.3. Facts describing the current state of the environment and agent actions are read from the tuple space, and when the precondition of a norm becomes true in the current environment, a norm instance (an obligation or a prohibition with a specified subject, deadline and sanction) is written into the tuple space. The subject agent receives a notification from the tuple space and retrieves the new norm.

2.2 N-2APL Multi-Agent System

N-2APL [3] is an extension of 2APL [8] with support for normative concepts including obligations, prohibitions, sanctions, deadlines and durations.

2APL is a BDI-based agent programming language that allows the implementation of agents in terms of cognitive concepts such as beliefs, goals and plans. A 2APL agent program specifies an agent's initial beliefs, goals, plans, and the reasoning rules it uses to select plans (`PG-rules`), to respond to messages and events (`PC-rules`), and to repair plans whose executions have failed (`PR-rules`). The initial beliefs of an

agent includes the agent's information about itself and its surrounding environment. The initial goals of an agent consists of formulas each of which denotes a situation the agent wants to realize (not necessarily all at once). The initial plans of an agent consists of tasks that an agent should initially perform.

In order to achieve its goals, an 2APL agent adopts plans. A plan consists of basic actions composed by sequence, conditional choice, conditional iteration and non interleaving operators. The non interleaving operator, $[\pi]$ where π is a plan, indicates that π is an *atomic* plan, i.e., the execution of π should not be interleaved with the execution of any other plan. Basic actions include external actions (which change the state of the agent's environment); belief update and goal adopt actions (which change the agent's beliefs and goals), and abstract actions (which provide an abstraction mechanism similar to procedures in imperative programming).

Planning goal rules allow an agent to select an appropriate plan given its goals and beliefs. A planning goal rule $\langle pgrule \rangle$ consists of three parts: the head of the rule, the condition of the rule, and the body of the rule. The body of the rule is a plan that is generated when the head (a goal query) and the condition (a belief query) of the rule are entailed by the agent's goals and beliefs, respectively. Procedure call rules (`PC-rules`) are used to select plans to handle messages and external events and to select a plan for an abstract action. As with planning goal rules, a procedure call rule $\langle pcrule \rangle$ consists of three parts: a head, a belief condition, and a plan. The head of the rule is an atom $\langle atom \rangle$, which represents either a message, an event, or an abstract action. The belief condition indicates when a message, event or abstract action should result in the plan forming the body of the rule being added to the agent's plan base. Plan repair rules are used to revise plans whose execution has failed. A plan repair rule $\langle prrule \rangle$ consists of three parts: a head consisting of an abstract plan, a belief condition and a body which is also an abstract plan. A plan repair rule indicates that if the execution of the first action of a plan matching the head of the rule fails and the belief condition is true, then the failed plan may be replaced by an instance of the plan forming the body of the rule.

To support norm-aware agents, N-2APL extends some key constructs of 2APL and restricts or changes the semantics of others. We briefly summarise these changes below; for full details, including the operational semantics of N-2APL and how it supports norm-aware deliberation, see [3].

Beliefs, Goals and Events Beliefs in N-2APL are the same as in 2APL and consist of Horn clause expressions. Goals in 2APL may be conjunctions of positive literals. In N-2APL we restrict goals to single atoms and extend their syntax to include optional deadlines. A *deadline* is a real time value (expressed in milliseconds) that specifies the time by which the goal should be achieved. If no deadline is specified for a goal as part of the agent's program, we assume a deadline of infinity. Norms are communicated to the agent in the form of events. An obligation event, represented as $obligation(\iota, o, d, s)$, specifies the time d by which the obligation o must be discharged, i.e., its deadline, and the sanction, s, that will be applied if the obligation is not discharged by the deadline. A prohibition event, represented as $prohibition(\iota, p, s)$, specifies a prohibition p that must not be violated and the sanction s that will be applied if execution of the agent's plans violates the prohibition. Obligations are adopted as goals with a deadline corresponding to the deadline of the obligation. In addition we extend the state of the agent

to include prohibitions, which are represented by single atoms, and the agent's initial state is extended to include its initial prohibitions. Lastly, we assume the the programmer provides function $pref(x)$ where x is a goal or prohibition, that returns the priority of the goal or prohibition x. For non-normative goals, the priority corresponds to the importance of achieving the goal state. In the case of prohibitions and goals derived from obligations, the priority corresponds to the importance of avoiding the sanction that would be incurred if the corresponding norm is violated.

Actions & Plans The syntax of external actions is extended to list the expected post-conditions of the action, to allow the prohibitions violated by a plan to be determined. In N-2APL, non-atomic plans are the same as in 2APL. However in N-2APL we change the interpretation of the non interleaving operator: $[\pi]$ indicates that the execution of π should not be interleaved with the execution of other *atomic* plans (rather than not interleaved with the execution of *any* other plan as in 2APL). In N-2APL, atomic plans are assumed to contain basic actions that may interfere only with the basic actions in other atomic plans. For example, a plan that involves moving to a new location should not be interleaved with other plans that change the agent's location. However, external actions in different non-atomic plans are executed in parallel, rather than being interleaved as in 2APL. Lastly, we restrict the scope of the non interleaving operator such that non-atomic plans cannot contain atomic sub-plans, either directly or through the expansion of an abstract action, i.e., plans to achieve top-level goals are either wholly atomic or non-atomic.

PG-rules We extend the syntax of plans in the body of a PG rule to include an optional field specifying the time required to execute the plan proposed by the PG rule. For simplicity, we assume that the time required to execute each plan π is fixed and known in advance.

The syntax of N-2APL is shown in Figure 2 in EBNF notation. Programming constructs in bold are exactly the same as in 2APL, and are omitted due to lack of space. For details, please see [8].

Our implementation of N-2APL was based on the implementation of 2APL developed at the University of Utrecht.[5] The extensions to the 2APL interpreter can be split into three main parts: modification of parser, extension of the agent's state to include obligations and prohibitions, and changes to agent's deliberation strategy. The 2APL parser is implemented using JavaCC, and the modifications necessary to accommodate the extended N-2APL syntax simply required changing the grammar specification. Obligation goals are stored in the existing 2APL goal base, and the original 2APL `Goal` class was extended to incorporate a deadline and a priority. Obligation deadlines are treated as relative times in milliseconds and transformed to goal deadlines (clock times) when the program in parsed (in the case of initial obligations) or when the obligation event is received from the normative organization. Prohibitions do not map to existing 2APL intentional attitudes. A specific prohibition base was therefore added to record the agent's current prohibitions.

Significant changes to the 2APL deliberation strategy were required to take the priorities and deadlines of goals and prohibitions into account when deliberating about

[5] The 2APL platform is available from apapl.sourceforge.net.

```
⟨Agent_Prog⟩     =  [ "Beliefs:" { ⟨belief⟩ } ],
                    [ "Goals:" ⟨goals⟩ ],
                    [ "Prohibitions:" ⟨prohibitions⟩ ],
                    [ "Plans:" ⟨plans⟩ ],
                    [ "PG-rules:" { ⟨pgrule⟩ } ],
                    [ "PC-rules:" { ⟨pcrule⟩ } ]
                    [ "PR-rules:" { ⟨prrule⟩ } ]
                    [ "Preferences:" ⟨prefs⟩ ]
⟨goals⟩          =  ⟨goal⟩ { ","⟨goal⟩ } ;
⟨goal⟩           =  ⟨atom⟩ ":" ⟨deadline⟩ ;
⟨prohibitions⟩   =  ⟨prohibition⟩ { ","⟨prohibition⟩ } ;
⟨prohibition⟩    =  ⟨atom⟩;
⟨pgrule⟩         =  ⟨goalquery⟩ "<-" ⟨belquery⟩ "|" ⟨plan⟩ ":" ⟨duration⟩;
⟨goalquery⟩      =  ⟨goalquery⟩ "and" ⟨goalquery⟩ | ⟨goalquery⟩
                    "or" ⟨goalquery⟩ | "(" ⟨goalquery⟩ ")" | ⟨atom⟩ ;
⟨belquery⟩       =  ⟨belquery⟩ "and" ⟨belquery⟩ | ⟨belquery⟩ "or" ⟨belquery⟩
                    | "(" ⟨belquery⟩ ")" | ⟨literal⟩ ;
⟨plan⟩           =  ⟨atomic-plan⟩ | ⟨non-atomic-plan⟩;
⟨atomic-plan⟩    =  "["⟨non-atomic-plan⟩"]";
⟨prefs⟩          =  ⟨pref⟩ { ","⟨pref⟩ } ;
⟨pref⟩           =  (⟨goal⟩ | ⟨sanction⟩) "->" ⟨priority⟩;
⟨sanction⟩       =  ⟨atom⟩;
⟨deadline⟩       =  ⟨time⟩;
⟨duration⟩       =  ⟨int⟩;
⟨priority⟩       =  ⟨int⟩;
```

Fig. 2. EBNF syntax of N-2APL

which plan to adopt for a goal and when to execute the plans to which it currently committed. The N-2APL deliberation strategy returns a schedule. A schedule is an assignment of a start or next execution time to a set of plans which ensures that: all plans complete by their deadlines, at most one atomic plan executes at any given time, and where the goals achieved and the prohibitions avoided are of the highest priority. Scheduling in N-2APL is pre-emptive in that the adoption of a new plan π may prevent previously scheduled plans with priority lower than π (including currently executing plans) being added to the new schedule. Plans that would exceed their deadline are dropped. In the case of obligations, a sanction will necessarily be incurred, so it is not rational for the agent to continue to attempt to discharge the obligation. In the case of goals, it is assumed that the deadline is hard, and there is no value in attempting to achieve the goal after the deadline. The deliberation strategy was modified so that after application of `PG-rules`, the set of previously scheduled and newly generated plans are scheduled, and plans with a scheduled next execution time of 'now' are then executed.

Changes were also required to the execution of atomic plans. To allow the interleaved execution of an atomic plan with non-atomic plans (rather than executing all the steps of an atomic plan in a single step as in 2APL), atomic plans are transformed into sequence plans during parsing of the agent's program code and flagged as being atomic. The plan execution module was also changed so that external actions in non-atomic plans are executed in parallel.

2.3 Tuple Space

Interaction between the 2OPL normative organization and the N-2APL agents is via a tuple space. We choose a tuple space rather than message-based interaction primarily to facilitate the integration of non-agent-based components such as the GeoSense game server (see Section 3). The facts recording the current (brute) state of the multi-agent environment and the detached norms and sanctions comprising its normative state are represented as tuples. The agents are connected to the tuple space through an extension of the N-2APL `Environment` class and in an agent program the tuple space is accessed in the same way as any other external environment.[6] The normative organisation accesses the tuple space through Prolog queries that wrap native Java method calls to the 'Prolog to Java' middleware used by both the N-2APL `Environment` class and 2OPL (see Figure 6).

The tuple space implementation is based on Jini JavaSpaces (Apache River). We choose JavaSpaces because of its simplicity and versatility [9], and because, like 2OPL and N-2APL, it is implemented in Java. JavaSpaces supports following primitive operations:

– `write` - writes a new entry into the tuple space.
– `read` - reads any matching entry from the space, blocking until one exists. Returns null if the timeout expires.
– `readIfExists` - reads any matching entry from the space, returning null if there is currently is none. Matching and timeouts are done as in read, except that blocking in this call is done only if necessary to wait for transactional state to settle.
– `notify` - when entries are written that match this template notify the given listener with a `RemoteEvent` that includes a handback object.
– `take` - take a matching entry from the space, waiting until one exists.

Both the normative organization and the multi-agent system synchronize with the tuple space. Using the `notify` method, the organization and the agents register to be notified when new a tuple matching a template is inserted in tuple space. For example, agents register to receive notifications about all new obligation and prohibition entries assigned to them, and the normative organization registers to receive notifications when a new agent location tuple is created in the tuple space. As the agents and the normative organisation receive only those updates that are relevant to them, the overhead of the tuple space relative to a message passing implementation is minimal.

Tuples are stored as serialized Java `Entry` objects. Each type of tuple is defined as a class that implements the `Entry` interface, and we defined a simple mapping from the Prolog terms used by 2OPL and N-2APL to `Entry` objects. JavaSpaces is non-deterministic and therefore all tuples need to be timestamped. Timestamps are implemented using a clock process which writes the current system time as a clock tuple in the tuple space. (In the example application described in the next section, the clock process is provided by the gameserver middleware, which writes a new clock tuple once a second.)

[6] To simplify the implementation, in the current prototype the effects (postcondition) of agent actions are written directly to the tuplespace, and 2OPL fact update rules are not used. However it would be straightforward to delegate action execution to 2OPL.

3 Example Application

To illustrate the flexibility of our framework, in this section we briefly describe its application in a novel normative application, a mixed reality game called GeoSense [6]. GeoSense is a real-time location-based game based on the MapAttack! game framework.[7] The game involves the use of GPS enabled smart phones to record the locations of players and display it on a game map. Players must reach specific physical locations within the game area to complete tasks and win the game. GeoSense is normally played by teams of human players. The long term objective of integrating our normative programming framework with the game is to investigate the use of norms as means of coordinating human-agent interaction in human agent collectives — systems which involve both human and agent participants. We envisage a system in which mixed teams of humans and agents play the game and the expected and prohibited behaviors of both human and agent participants are expressed in terms of norms. However the version of the application described below involves only software agents.

Fig. 3. GeoSense web interface

The GeoSense game is played on a map of a physical location (typically part of a city such as a park) and has three kinds of players: a truck, pursuers and coordinators. The truck carries a load of radioactive waste, and attempts to avoid detection. The pursuers, assisted by the coordinator(s), attempt to determine the location of the truck. The physical (GPS) locations of the pursuers are shown on the map and updated as the players move in the real environment. The pursuers' locations are visible to each other and to the coordinator(s). The truck is a virtual player, and its location is not visible on the map. However its radioactive load leaves a virtual 'trace' that can be measured by by taking a 'reading' at a pursuer's current physical location. The reading ranges from

[7] mapattack.org

0 to 100, with higher readings indicating a smaller distance to the truck. In an attempt to avoid detection, the truck may drop some of its load as it moves through the game area. Such dropped waste also gives a positive reading, making it more difficult for the pursuers to determine the location of truck. The coordinator(s) have a global view of the positions of all the pursuers and of all recent readings. The role of the coordinator is to aid the pursuers by directing them to promising areas of the map. The coordinator can request that a pursuer takes reading at a particular physical location by placing a virtual 'coin' at the location on the map. The pursuer must then go the physical location indicated by the coin and take a reading.

GeoSense is written in Ruby and runs as a web server. Clients can connect to the server through `HTTP` or `Socket.IO` interfaces. Clients are either a mobile device for a pursuer or a web browser for a coordinator. The web interface of the game is illustrated in Figure 3.

3.1 Encoding Game Rules as Norms

The rules of the GeoSense game are encoded in the gameserver code and are not accessible to agents. To allow agents to participate in the game, we re-expressed the GeoSense game rules as a set of 2OPL obligations and prohibitions. The norms specify which game states the agents should try to bring about (and by when) or are prohibited from bringing about, and any sanction incurred if the norm is violated, e.g., a deduction in points. For example, a norm may specify that the truck is prohibited from entering a particular area of the map, and that violation of the norm results in the loss of 500 points. (Note that a norm-aware agent may still choose to violate a norm e.g., the agent may enter a prohibited area if doing so allows it to win the game.) Updates to the game state resulting from agent actions may trigger norms that apply to the agent that performed the action or another agent. For example, when a coordinator places a coin for a pursuer, the normative organisation creates an obligation that the pursuer must take a reading at the location of the coin within a specified time, and a prohibition specifying that the coordinator cannot place another coin at the same location. Example 2OPL game norms are illustrated in Figure 4.

```
norm(forbidden_area(Agent),
     (truck(Agent), forbidden(X,Y)),
     prohibition(Agent, [at(X,Y,Agent)],
                 [reduce_score(Agent,500)])
).

norm(take_reading(Agent),
     (pursuer(Agent), coin(X,Y,Agent), clock(Now)),
     obligation(Agent, [reading(X,Y,Agent)],
                Now + 15000, [reduce_score(Agent,300)])
).
```

Fig. 4. Example GeoSense game norms

3.2 Agent Programs

We also developed N-2APL programs to allow the agents to play the game and achieve the goals resulting from the game norms. As an example, a program for a simple truck agent is shown in Figure 5. The truck has two goals. The first goal `at(2,2) : 120000` is to reach position `(2,2)` in 2 minutes (120,000 msecs) from the start of the game and has a priority of 3. The second goal `dropLoad : 60000` is to drop (part of) its load within one minute of the start of the game, and has a priority of 5. When the agent adopts a goal it executes the matching PG-rule. For example, the rule to achieve the `at(X,Y)` goal specifies a plan that involves moving to the required position. The PG-rule also includes an estimate of the time required to execute the plan (one minute in this case).

```
Beliefs:
  points(1000).
  position(19,19).
  clock(0).

Goals:
  at(2,2) : 120000,
  dropLoad : 60000

Preferences:
  at(2,2) -> 3,
  reduce_score(truck, 500) -> 4,
  dropLoad -> 5

PG-rules:

at(X,Y) <- true | { moveTo(X,Y); } : 60000
dropLoad <- position(10,10) | { drop(X,Y); } : 1000
```

Fig. 5. N-2APL program for the Truck agent

The obligations and prohibitions the agent receives as a result of the game rules may conflict with its own goals in the game. For example, the agent's goals to be at `(2,2)` or to drop part of its load when at position `(10,10)` may require visiting a prohibited area of the map. In such a situation, a norm-aware agent must choose between its existing goals and the norms imposed by the game. Critically, a N-2APL agent is able to violate norms (accepting the resulting sanctions) if it is in the agent's overall interests to do so. For example, the truck agent assigns a higher priority to achieving the goal `at(2,2)` than to the sanction resulting from entering the prohibited area (losing 500 points), which in turn has a higher priority than the `dropLoad` goal. The agent will therefore enter the prohibited area if it necessary to reach `(2,2)` but would not violate the norm to drop part of its load.

3.3 Gameserver Integration

To maintain the game state (and allow future participation by human players using the GeoSense mobile and web browser clients), we integrated the GeoSense gameserver with our normative programming framework consisting of 2OPL, N-2APL and JavaSpaces. GeoSense is connected to the framework through the tuple space as shown in Figure 6. Updates to the tuple space corresponding to player actions are converted to HTTP `POST` requests to the game server. For example when a pursuer agent updates its location, the move action adds a new tuple to the tuple space, which is sent as a `POST` request to the gameserver. Similarly, the JSON updates generated by the gameserver used by the smart phone mobile clients are converted into tuples in the tuple space.

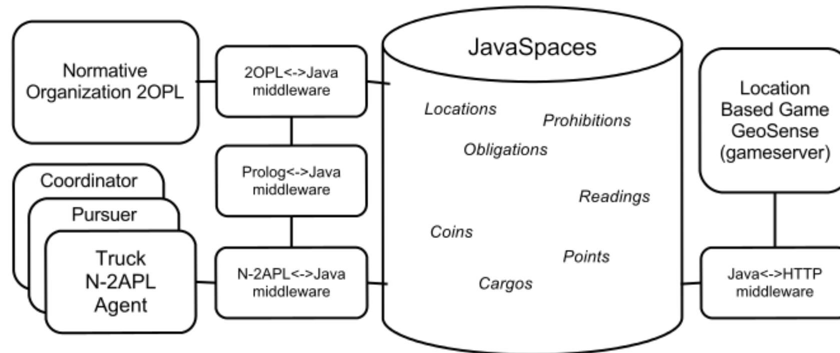

Fig. 6. Overall system architecture

To simplify development of the agent programs, the tuple space to HTTP middleware discretises some aspects of the game state. For example, the locations of the players are represented as longitude and latitude pairs by the gameserver, while the agents see the game environment as a grid and move one cell at a time. Similarly, the real-time clock used by the gameserver to record the progress of the game is seen as a series of one second ticks by the agents. However these simplifications do not affect game play and are not inherent in the normative programming framework itself.

The agents' beliefs and actions are synchronized with the game state via the tuple space, allowing them to participate in the game. Moreover the actions of the agents are coordinated and regulated through the norms that implement the game rules.

4 Related Work

There has been considerable work on normative programming frameworks and middleware to support the development of normative multi-agent organisations, and such

frameworks are often designed to inter-operate with existing BDI-based agent programming languages. However in these frameworks, the agents do not deliberate about whether to comply with norms.

For example, $\mathcal{J}$-$\mathcal{M}$OISE$^+$ [10] is designed to inter-operate with the $\mathcal{S}$-$\mathcal{M}$OISE$^+$ [11] middleware and allows Jason [12] agents to access and update the state of an $\mathcal{S}$-$\mathcal{M}$OISE$^+$ organization. Similarly, the JaCaMo programming framework combines the Jason, Cartago [13], and $\mathcal{S}$-$\mathcal{M}$OISE$^+$ platforms. In JaCaMo, the organisational infrastructure of a multiagent system consists of organisational artefacts and agents that together are responsible for the management and enactment of the organisation. JaCaMo provides similar functionality to $\mathcal{J}$-$\mathcal{M}$OISE$^+$ in allowing Jason agents to interact with organisational artefacts, e.g., to take on a certain role. However while these approaches allow a developer to program e.g., when an agent should adopt a role, the Jason agents have no explicit mechanisms to reason about norms and their deadlines and sanctions in order to adapt their behaviour at run time. Another approach that integrates norms in a BDI-based agent programming architecture is proposed in [14]. This extends the AgentSpeak(L) architecture with a mechanism that allows agents to behave in accordance with a set of non-conflicting norms. As in N-2APL, the agents can adopt obligations and prohibitions with deadlines, after which plans are selected to fulfil the obligations or existing plans are suppressed to avoid violating prohibitions. However, unlike N-2APL, [14] does not consider scheduling of plans with respect to their deadlines or possible sanctions.

In contrast to frameworks such as $\mathcal{S}$-$\mathcal{M}$OISE$^+$ [11] which regulate behaviour by norm regimentation, our approach is based on norm enforcement and sanctions. Frameworks such as ORA4MAS [15] provide support for both norm regimentation and enforcement, however monitoring must be explicitly coded in organizational artifacts. An advantage of using a tuple space to represent both the brute and normative state of the agent's environment is monitoring of norm compliance and violation by the 2OPL interpreter is greatly simplified. On the other hand, approaches such as ORA4MAS allow decentralized (and arguably more flexible) decision making about the appropriate sanction for a violation.

A number of normative programming languages have recently been proposed that are similar in spirit to the 2OPL language used in our framework. NPL/NOPL [16] allows the expression of norms with conditions, obligations and deadlines, and norms may be regimented or enforced. However sanctions are represented as an obligation that an agent apply the sanction to the agent that violated the norm, whereas in our framework sanctions are applied by the organization. The norm-oriented language proposed in [17] is rule based like 2OPL. However, their norms relate to actions the agents should or should not perform while 2OPL norms relate to a state of the environment that should (or should not) be brought about. The normative language of the THOMAS multi-agent architecture [18] supports conditional norms with deadlines, sanctions and rewards. Conditions refer to actions (and optionally states). Norms are enforced rather than regulated, and sanctions may be applied by agents rather than the organization. As in our approach, the normative infrastructure does not restrict interactions between agents. A rule-based system implemented in Jess maintains a fact base representing the organizational state, detects norm activation and monitors violations. While these ap-

proaches offer similar functionality to 2OPL and the tuple space in our framework, they have not been integrated with a norm-aware agent programming language.

There has been relatively little work on applying norms to games. In [19] the authors describe the use of of expectation monitoring by agents in the Second Life virtual environment. An expectation monitoring component integrated into the Jason interpreter allows agents to detect fulfilment and violation of their expectations. Expectations have some similarities to norms in specifying conditional constraints on future states. However, they are local to an agent rather than generated by a normative organization and there is no centralized monitoring or sanctioning of agents that violate expectations. Moreover, while the approach described in [19] allows agents to detect violations of expectations without recourse to a normative organization, the issues of how expectations are generated and what to do when they are fulfilled or violated are left to the agent developer. Perhaps the work that is most similar to ours is [20], in which the $\mathcal{M}$OISE^{inst} normative organisation meta-model is used to control an interactive TV game show in which the avatars are implemented as agents. The purpose of the norms is to constrain players and their avatars to adopt team behaviour and to respect rules, while allowing some autonomy.

5 Conclusions and Future Work

We described a framework for programming norm-aware multi-agent systems which integrates the N-2APL norm-aware agent programming language with the 2OPL language for programming normative organisations. To the best of our knowledge, this is the first implementation of N-2APL and the first implementation of an integrated framework for norm-aware multi-agent systems in which autonomous agents deliberate about whether to conform to the norms imposed by a normative organisation. To illustrate the flexibility of our framework, we described its application in a location-based mixed reality game called GeoSense. We showed how the game rules can be expressed as conditional norms with deadlines and sanctions, and how agents can deliberate about their individual goals and the norms imposed by the game.

The GeoSense game is normally played by teams of human players. In future work, we plan to use the integration of norm-aware agents and the GeoSense game to investigate the use of norms for coordinating interaction and achieving adjustable autonomy in systems involving both human and agent participants. We also plan to address some of the limitations of our current implementation. For example, our approach currently assumes that the normative organisation assigns norms and sanctions to individual agents. While this is appropriate for many applications, there are situations where it would be more natural to address norms and sanctions to a group of agents. For example, a coordinator agent may create an obligation that some pursuer agent take a reading at a particular location without specifying which agent should do so; if none of the agents discharge the obligation by the deadline, the normative organisation applies a sanction to the pursuers as a group. In future work we plan to look at extending our framework to incorporate group norms and sanctions.

Acknowledgements

We would like to thank Wenchao Jiang for making the code of the GeoSense game available and for assistance in developing the gameserver middleware. This work was partially supported by EPSRC grant EP/I011587/1.

References

1. Dastani, M., Grossi, D., Meyer, J.J.C., Tinnemeier, N.: Normative multi-agent programs and their logics. In Meyer, J.J.C., Broersen, J., eds.: Knowledge Representation for Agents and Multi-Agent Systems, Berlin, Heidelberg, Springer-Verlag (2009) 16–31
2. Hübner, J.F., Boissier, O., Bordini, R.H.: From organisation specification to normative programming in multi-agent organisations. In Dix, J., Leite, J., Governatori, G., Jamroga, W., eds.: Proceedings of the 11th International Workshop on Computational Logic in Multi-Agent Systems (CLIMA XI). Volume 6245 of Lecture Notes in Computer Science., Lisbon, Portugal, Springer (2010) 117–134
3. Alechina, N., Dastani, M., Logan, B.: Programming norm-aware agents. In Conitzer, V., Winikoff, M., Padgham, L., van der Hoek, W., eds.: Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2012), Valencia, Spain, IFAAMAS (2012) 1057–1064
4. Castelfranchi, C., Dignum, F., Jonker, C.M., Treur, J.: Deliberative normative agents: Principles and architecture. In Jennings, N.R., Lesperance, Y., eds.: Intelligent Agents VI. Proceedings of the 6th International Workshop on Agent Theories and Architectures and Languages and ATAL'99. LNAI, Springer Verlag (2000) 364–378
5. van Riemsdijk, M.B., Hindriks, K.V., Jonker, C.M.: Programming organization-aware agents: A research agenda. In: Proceedings of the Tenth International Workshop on Engineering Societies in the Agents' World (ESAW'09). Volume 5881 of LNAI., Springer (2009) 98–112
6. Fischer, J.E., Jiang, W., Moran, S.: AtomicOrchid: A mixed reality game to investigate coordination in disaster response. In Herrlich, M., Malaka, R., Masuch, M., eds.: Entertainment Computing - ICEC 2012. Volume 7522 of Lecture Notes in Computer Science., Springer Berlin Heidelberg (2012) 572–577
7. Tinnemeier, N.: Organizing Agent Organizations: Syntax and Operational Semantics of an Organization-Oriented Programming Language. PhD thesis, Utrecht University, SIKS (2011)
8. Dastani, M.: 2APL: A practical agent programming language. Autonomous Agents and Multi-Agent Systems **16** (2008) 214–248
9. Oaks, S., Wong, H.: Jini in a nutshell - a desktop quick reference. O'Reilly (2000)
10. Hübner, J.F., Sichman, J.S., Boissier, O.: Developing organised multi-agent systems using the $\mathcal{M}$OISE$^+$ model: Programming issues at the system and agent levels. International Journal of Agent-Oriented Software Engineering **1** (2007) 370–395
11. Hübner, J., Sichman, J., Boissier, O.: $\mathcal{S}$-$\mathcal{M}$OISE$^+$: A middleware for developing organised multi-agent systems. In: Proceedings of the International Workshop on Coordination, Organizations, Institutions, and Norms in Multi-Agent Systems. Volume 3913 of LNCS., Springer (2006) 64–78
12. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming multi-agent systems in AgentSpeak using Jason. Wiley Series in Agent Technology. Wiley (2007)

13. Ricci, A., Viroli, M., Omicini, A.: Give agents their artifacts: the A&A approach for engineering working environments in MAS. In Durfee, E.H., Yokoo, M., Huhns, M.N., Shehory, O., eds.: Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2007), IFAAMAS (2007) 601–603
14. Meneguzzi, F.R., Luck, M.: Norm-based behaviour modification in BDI agents. In Sierra, C., Castelfranchi, C., Decker, K.S., Sichman, J.S., eds.: Proceedings of the 8th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009), IFAAMAS (2009) 177–184
15. Hübner, J., Boissier, O., Kitio, R., Ricci, A.: Instrumenting multi-agent organisations with organisational artifacts and agents. Autonomous Agents and Multi-Agent Systems **20** (2010) 369–400
16. Hübner, J., Boissier, O., Bordini, R.: A normative programming language for multi-agent organisations. Annals of Mathematics and Artificial Intelligence **62** (2011) 27–53
17. García-Camino, A., Rodríguez-Aguilar, J., Sierra, C., Vasconcelos, W.: Constraint rule-based programming of norms for electronic institutions. Autonomous Agents and Multi-Agent Systems **18** (2009) 186–217
18. Criado, N., Julin, V., Botti, V., Argente, E.: A norm-based organization management system. In Padget, J., Artikis, A., Vasconcelos, W., Stathis, K., Silva, V., Matson, E., Polleres, A., eds.: Coordination, Organizations, Institutions and Norms in Agent Systems V. Volume 6069 of Lecture Notes in Computer Science., Springer Berlin Heidelberg (2010) 19–35
19. Ranathunga, S., Cranefield, S., Purvis, M.: Integrating expectation monitoring into BDI agents. In Dennis, L., Boissier, O., Bordini, R., eds.: Proceedings of the Ninth International Workshop on Programming Multi-Agent Systems (ProMAS 2011). Volume 7217 of Lecture Notes in Computer Science., Springer Berlin Heidelberg (2012) 74–91
20. Gâteau, B., Boissier, O., Khadraoui, D., Dubois, E.: Controlling an interactive game with a multi-agent based normative organisational model. In Noriega, P., Vázquez-Salceda, J., Boella, G., Boissier, O., Dignum, V., Fornara, N., Matson, E., eds.: Coordination, Organizations, Institutions, and Norms in Agent Systems II, Berlin, Heidelberg, Springer-Verlag (2007) 86–100