Norm Refinement and Design through Inductive Learning*

Domenico Corapi[1], Marina De Vos[2], Julian Padget[2],
Alessandra Russo[1], and Ken Satoh[3]

[1] Department of Computing, Imperial College London
{d.corapi,a.russo}@imperial.ac.uk
[2] Department of Computer Science, University of Bath
{mdv,jap}@cs.bath.ac.uk
[3] National Institute of Informatics
ksatoh@nii.ac.jp

Abstract. In the physical world, the rules governing behaviour are debugged by observing an outcome that was not intended and the addition of new constraints intended to prevent the attainment of that outcome. We propose a similar approach to support the incremental development of normative frameworks (also called institutions) and demonstrate how this works through the validation and synthesis of normative rules using model generation and inductive learning. This is achieved by the designer providing a set of *use cases*, comprising collections of event traces that describe how the system is used along with the desired outcome with respect to the normative framework. The model generator encodes the description of the current behaviour of the system. The current specification and the traces for which current behaviour and expected behaviour do not match are given to the learning framework to propose new rules that revise the existing norm set in order to inhibit the unwanted behaviour. The elaboration of a normative system can then be viewed as a semi-automatic, iterative process for the detection of incompleteness or incorrectness of the existing normative rules, with respect to desired properties, and the construction of potential additional rules for the normative system.

1 Introduction

Norms and regulations play an important role in the governance of human society. Social rules such as laws, conventions and contracts prescribe and regulate our behaviour, however it is possible for us to break these rules at our discretion and face the consequences. By providing the means to describe and reason about norms in a computational context, normative frameworks (also called institutions or virtual organisations) may be applied to software systems allowing for automated reasoning about the consequences of socially acceptable and unacceptable behaviour, by monitoring the permissions, empowerment and obligations of the participants and generating violations when norms are not followed.

* This work is partially supported through the EU Framework 7 project *ALIVE (FP7-IST-215890)*, and the EPSRC PRiMMA project (EP/F023294/1).

The formal model put forward in [9] and its corresponding operationalisation through Answer Set Programming (ASP) [3, 18] aims to support the top-down design of normative frameworks. $AnsProlog$ is a knowledge representation language that allows the programmer to describe a problem and required properties on the solutions in an intuitive way. Programs consist of rules interpreted under the answer set semantics. Answer set solvers, like CLASP[17] or SMODELS[25], can be used to reason about the given $AnsProlog$ specification, by returning *acceptable solutions* in the form of traces, as answer sets. In a similar way, the correctness of the specification with respect to given properties can be verified.

Currently, the elaboration of behavioural rules and norms is an error-prone process that relies on the manual efforts of the designer and would, therefore, benefit from automated support. In this paper, we present an inductive logic programming (ILP) [24] approach for the extraction of norms and behaviour rules from a set of use cases. The approach is intended as a design support tool for normative frameworks. Complex systems are hard to model and even if testing of properties is possible, sometimes it is hard to identify missing or incorrect rules. In some cases, e.g. legal reasoning, the abstract specification of the system can be in part given in terms of specific instances and use cases that ultimately drive the design process and are used to assess it. We propose a design support tool that employs *use-cases*, i.e. traces together with their expected normative behaviour, to assist in the revision of a normative framework. The system is correct when none of the traces are considered *disfunctional*, i.e. they match the expected normative behaviour. When a disfunctional trace is encountered the normative specification needs to be adjusted: the task is to refine the given description by learning missing norms and/or behavioural rules that, added to the description, entail the expected behaviour over the traces. We show how this task can be naturally represented as a non-monotonic ILP problem in which the partial description of the normative system provides the background knowledge and the expected behaviour comprises the examples. In particular, we show how a given $AnsProlog$ program and traces can be reformulated into an ILP representation that makes essential use of negation in inducing missing parts of the specification. As the resulting learning problem is inherently non-monotonic, we use a non-monotonic ILP system, called TAL [12], to compute the missing specification from the traces and the initial description.

Given the declarative nature of ASP, the computational paradigm used for our normative frameworks, we needed to adopt a declarative learning approach as we aim to learn declarative specifications. This differs from other approaches, such as reinforcement learning whereby norms or policies are learned as outcomes of estimation and optimisation processes. Such types of policies are not directly representable in a declarative format and are quite different in nature from the work reported here.

The paper is organised as follows. Section 2 presents some background material on the normative framework, while Section 3 introduces the non-monotonic ILP system used in our proposed approach. Section 4 describes the $AnsProlog$ modelling of normative frameworks. Section 5 illustrates how the revision task can be formulated into an ILP problem, and how the generated ILP hypothesis can be reformulated as norms and behaviour rules within the $AnsProlog$ representation. In Section 6 we illustrate the flexibility and expressiveness of our approach through a number of different par-

tial specifications of a reciprocal file sharing normative framework. Section 7 relates our approach to existing work on learning norms with respects to changing/improved requirements. We conclude with a summary and remarks about future work.

2 Normative Frameworks

The concept of normative framework has become firmly embedded in the agent community as a necessary foil to the essential autonomy of agents, in just the same way as societal conventions and legal frameworks have grown up to constrain people. In both the physical and the virtual worlds, and the emerging combination of the two, the arguments in favour centre on the minimisation of disruptive behaviour and supporting the achievement of the goals for which the normative framework has been conceived and thus also the motivation for submission to its governance by the participants. While the concept remains attractive, its realisation in a computational setting remains a subject for research, with a wide range of existing logics [29, 1, 7, 9, 32] and tools [26, 14, 19].

2.1 Formal Model

To provide context for this paper, we give an outline of a formal event-based model for the specification of normative frameworks that captures all the essential properties, namely empowerment, permission, obligation and violation. Extended presentations appear in [9] and [10].

The essential elements of our normative framework are: (i) events ($\mathcal{E}$), that bring about changes in state, and (ii) fluents ($\mathcal{F}$), that characterise the state at a given instant. The function of the framework is to define the interplay between these concepts over time, in order to capture the evolution of a particular institution through the interaction of its participants. We distinguish two kinds of events: normative events ($\mathcal{E}_{norm}$), that are the events defined by the framework and exogenous ($\mathcal{E}_{ex}$), that are outside its scope, but whose occurrence triggers normative events in a direct reflection of the "counts-as" principle [21]. We further partition normative events into normative actions ($\mathcal{E}_{act}$) that denote changes in normative state and violation events ($\mathcal{E}_{viol}$), that signal the occurrence of violations. Violations may arise either from explicit generation, from the occurrence of a non-permitted event, or from the failure to fulfil an obligation. We also distinguish two kinds of fluents: *normative fluents* that denote normative properties of the state such as permissions $\mathcal{P}$, powers $\mathcal{W}$ and obligations $\mathcal{O}$, and *domain fluents* $\mathcal{D}$ that correspond to properties specific to the normative framework itself. The set of all fluents is denoted as $\mathcal{F}$. A normative state is represented by the fluents that hold true in this state. Fluents that are not presented are considered to be false. Conditions on a state are therefore expressed by a set of fluents that should be true or false. The set of possible conditions is referred to as $\mathcal{X} = 2^{\mathcal{F} \cup \neg \mathcal{F}}$.

Changes in state are achieved through the definition of two relations: (i) the generation relation, which implements counts-as by specifying how the occurrence of one (exogenous or normative) event generates another (normative) event, subject to the empowerment of the actor and the conditions on the state, and (ii) the consequence relation. This latter specifies the initiation and termination of fluents subject to the performance of some action in a state matching some expression. The generation relation is formally

defined as $\mathcal{G} : \mathcal{X} \times \mathcal{E} \rightarrow 2^{\mathcal{E}_{norm}}$, and the consequence relation as $\mathcal{C} : \mathcal{X} \times \mathcal{E} \rightarrow 2^{\mathcal{F}} \times 2^{\mathcal{F}}$. The fluents to be initiated as a result of an event E are often denoted by $\mathcal{C}^{\uparrow}(\phi, E)$ while the ones to be terminated are denoted by $\mathcal{C}^{\downarrow}(\phi, E)$.

The semantics of our normative framework is defined over a sequence, called a *trace*, of exogenous events. Starting from the initial state, each exogenous event is responsible for a state change, through initiation and termination of fluents. This is achieved by a three-step process: (i) the transitive closure of $\mathcal{G}$ with respect to a given exogenous event determines all the generated (normative) events, (ii) to this all violations of events not permitted and obligations not fulfilled are added, giving the set of all events whose consequences determine the new state, (iii) the application of $\mathcal{C}$ to this set of events identifies all fluents that are initiated and terminated with respect to the current state so giving the next state. For each trace, we can therefore compute a sequence of states that constitutes the model of the normative framework for that trace. This process is realised as a computational model through Answer Set Programming (see Section 4) and it is this representation that is the subject of the learning process described in Section 5.

3 Learning

Inductive Logic Programming (ILP) [24] is a machine learning technique concerned with the induction of logic theories from (positive and negative) examples and has been successfully applied to a wide range of problems [15]. Automatic induction of hypotheses represented as logic programs is one of the distinctive features of ILP. Moreover, the use of logic programming as representation language allows a principled representation of background information relevant to the learning. To refine normative theories we employ an ILP learning system, called TAL [12], that is able to learn non-monotonic theories, and can be employed to perform learning of new rules and the revision of existing rules. The TAL approach is based on mapping a given inductive problem into an abductive reasoning process. The current implementation of TAL relies on an extension of the abductive procedure SLDNFA [13] and preserves its semantics.

Definition 1. *A* non-monotonic ILP task *is defined as* $\langle E, B, S \rangle$ *where* E *is a set of ground positive or negative literals, called* examples*,* B *is a* background *normal theory and* S *is a set of clauses called* language bias*. The normal theory* $H \in ILP\langle E, B, S \rangle$*, called* hypothesis*, is an inductive solution for the task* $\langle E, B, S \rangle$*, if* $H \subseteq S$*,* H *is consistent with* B *and* $B \cup H \models E$*.*

B and H are normal theories and thus support negation as failure. The choice of an appropriate language bias is critical. In TAL the language bias S is specified by means of *mode declarations* [**?**].

Definition 2. *A* mode declaration *is either a head or body declaration, respectively* $modeh(s)$ *and* $modeb(s)$ *where* s *is called a* scheme*. A scheme* s *is a ground literal containing place-markers. A* place-marker *is a ground function whose functor is one of the three symbols '+' (input), '-' (output), '#' (constant) and the argument is a constant called* type*.*

Given a schema s, s^* is the literal obtained from s by replacing all place-markers with different variables $X_1, ..., X_n$. A rule r is *compatible* with a set M of mode declarations iff (a) there is a mapping from each head/body literal l in r to a head/body declaration $m \in M$ with schema s such that each literal is subsumed by its corresponding s^*; (b) each output place-marker is bound to an *output variable*; (c) each input place-marker is bound to an output variable appearing in the body or to a variable in the head; (d) every constant place-marker is bound to a constant; (e) all variables and constants are of the corresponding type. From a user perspective, mode declarations establish how rules in the final hypotheses are structured, defining literals that can be used in the head and in the body of a well-formed hypothesis. Although we show M in the running example of this paper for reference, the mode declarations can be concealed from the user and derived automatically. They can be optionally refined to constrain the search whenever the designer wants to employ useful information on the outcome of the learning to reduce the number of alternative hypotheses or improve performance.

4 Modelling Normative Frameworks

While the formal model of a normative framework allows for clear specification of a normative system, it is of little support to designers or users of these systems. In order to be able to do so, computational tools are needed. The first step is a computational model equivalent to the formal model. We have opted for a form of logic programming, called Answer Set Programming (ASP)[18]. Here we only present a short flavour of the language $AnsProlog$, and the interested reader is referred to [3] for in-depth coverage.

$AnsProlog$ is a knowledge representation language that allows the programmer to describe a problem and the requirements on the solutions in an intuitive way, rather than the algorithm to find the solutions to the problem. The basic components of the language are atoms, elements that can be assigned a truth value. An atom can be negated using *negation as failure* so creating the *literal* $\mathrm{not}\ a$. We say that $\mathrm{not}\ a$ is true if we cannot find evidence supporting the truth of a. If a is true then $\mathrm{not}\ a$ is false and vice versa. Atoms and literals are used to create rules of the general form: $a \leftarrow B, \mathrm{not}\ C$, where a is an atom and B and C are set of atoms. Intuitively, this means *if all elements of B are known/true and no element of C is known/true, then a must be known/true*. We refer to a as the head and $B \cup \mathrm{not}\ C$ as the body of the rule. Rules with empty body are are called *facts; A* program in $AnsProlog$ is a finite set of rules.

The semantics of $AnsProlog$ are defined in terms of *answer sets*, i.e. assignments of true and false to all atoms in the program that satisfy the rules in a minimal and consistent fashion. A program has zero or more answer sets, each corresponding to a solution.

4.1 Mapping the formal model into $AnsProlog$

In this section we only provide a summary description of how the formal institutional model is translated in to $AnsProlog$. A full description of the model can be found in [9] together with completeness and correctness of model with respect to traces. Each program models the semantics of the normative framework over a sequence of n time

instants such that $t_i : 0 \leq i \leq n$. Events are considered to occur *between* these snapshots, where for simplicity we do not define the intervals at which events occur explicitly, and instead refer to the time instant at the start of the interval at which an event is considered to occur. Fluents may be true or false at any given instant of time, so we use atoms of the form $\texttt{holdsat}(\texttt{f}, \texttt{t}_\texttt{i})$ to indicate that fluent $\texttt{f}$ holds at time instant $\texttt{t}_\texttt{i}$. In order to represent changes in the state of fluents over time, we use atoms of the form $\texttt{initiated}(\texttt{f}, \texttt{t}_\texttt{i})$ and $\texttt{terminated}(\texttt{f}, \texttt{t}_\texttt{i})$ to denote the fact that fluent $\texttt{f}$ was initiated or terminated, respectively, *between* time instants i and $i+1$. We use atoms of the form $\texttt{occurred}(\texttt{e}, \texttt{t}_\texttt{i})$ to indicate that event $\texttt{e} \in \mathcal{E}$ is considered to have occurred between instant $\texttt{t}_\texttt{i}$ and $\texttt{t}_{\texttt{i}+1}$. These atoms denote events that occur in an external context or are generated by the normative framework. For exogenous events we additionally use atoms of the form $\texttt{observed}(\texttt{e}, \texttt{t}_\texttt{i})$ to denote the fact that e has been observed.

The mapping of a normative framework consists of three parts: a base component which is independent of the framework being modelled, the time model and the framework specific component. The independent component deals with inertia of the fluents, the generation of violation events of un-permitted actions and unsatisfied obligations. The time model defines the predicates for time and is responsible for generating a single observed event at every time instance. In this paper we will focus solely on the representation of the specific features of the normative framework.

In order to translate rules in the normative framework relations $\mathcal{G}$ and $\mathcal{C}$, we must first define a translation for expressions which may appear in these rules. The valuation of a given expression taken from the set $\mathcal{X}$ depends on which fluents may be held to be true or false in the current state (at a give time instant). We translate expressions into ASP rule bodies as conjunctions of extended literals using negation as failure for negated expressions.

With all these atoms defined, mapping the generation function and the consequence relation of a specific normative framework becomes rather straightforward. The generation function specifies that an normative event e occurs at a certain instance ($\texttt{occurred}(\texttt{e}, \texttt{t})$) when an another event e' occurs, the event e is empowered ($\texttt{holdsat}(\texttt{pow}(\texttt{e}), \texttt{t})$ and a set of conditions on the state are satisfied ($\texttt{holdsat}(\texttt{f}, \texttt{t})$ or $\textbf{not}\ \texttt{holdsat}(\texttt{f}, \texttt{t})$). The rules for initiation ($\texttt{initiated}(\texttt{f}, \texttt{t})$) and termination ($\texttt{terminated}(\texttt{f}, \texttt{t})$ of a fluent f are triggered when a certain event e occurs ($\texttt{occurred}(\texttt{e}, \texttt{t})$) and a set of conditions on the state are fulfilled. The initial state of our normative framework is encoded as simple facts ($\texttt{holdsat}(\texttt{f}, \texttt{i00})$).

Figure 1 gives a summary of all $AnsProlog$ rules that are generated for a specific normative framework, including the definition of all the fluents and events as facts. For a given expression $\phi \in \mathcal{X}$, we use the term $EX(\phi, T)$ to denote the translation of ϕ into a set of ASP literals of the form $\texttt{holdsat}(\texttt{f}, \texttt{T})$ or $\textbf{not}\ \texttt{holdsat}(\texttt{f}, \texttt{T})$.

In situations where the normative system consists of a number of agents whose actions can be treated in the same way (e.g. the rules for borrowing a book are the same for every member of alibrary) or where the state consists of fluents that can be treated in a similar way (e.g. the status of book), we can parameterise the events and fluents. This is represented in the $AnsProlog$ program by function symbols (e.g $\texttt{borrowed}(\texttt{Agent}, \texttt{Book})$) rather than terms. To allow for grounding, extra atoms to

$$
\begin{aligned}
p \in \mathcal{F} &\Leftrightarrow \texttt{ifluent(p)}.\\
e \in \mathcal{E} &\Leftrightarrow \texttt{event(e)}.\\
e \in \mathcal{E}_{ex} &\Leftrightarrow \texttt{evtype(e,obs)}.\\
e \in \mathcal{E}_{act} &\Leftrightarrow \texttt{evtype(e,act)}.\\
e \in \mathcal{E}_{viol} &\Leftrightarrow \texttt{evtype(e,viol)}.\\
\mathcal{C}^{\uparrow}(\phi, e) = P &\Leftrightarrow \forall p \in P \cdot \texttt{initiated(p,T)} \leftarrow \texttt{occurred(e,I)}, EX(\phi, T).\\
\mathcal{C}^{\downarrow}(\phi, e) = P &\Leftrightarrow \forall p \in P \cdot \texttt{terminated(p,T)} \leftarrow \texttt{occurred(e,I)}, EX(\phi, T).\\
\mathcal{G}(\phi, e) = E &\Leftrightarrow g \in E, \texttt{occurred(g,T)} \leftarrow \texttt{occurred(e,T)},\\
&\qquad\qquad \texttt{holdsat(pow(e),I)}, EX(\phi, T).\\
p \in S_0 &\Leftrightarrow \texttt{holdsat(p,i00)}.
\end{aligned}
$$

Fig. 1. The translation of normative framework specific rules into $AnsProlog$

ground these variables need to be added. Grounded versions of the atoms also need to be added to the program. An example of this can be found in Section 6.

5 Learning Normative Rules

5.1 Methodology

The development process is supported by a set of *use cases* U. Use cases represent instances of executions that are known to the designer and that drive the elaboration of the normative system. If the current formalisation of the system does not match the intended behaviour in the use case then the formalisation is still not complete or incorrect. Each use case $u \in U$ is a tuple $\langle T, C, O \rangle$ where T is a *trace* that specifies *all* the exogenous events occurring at all the time points considered (`observed(e,T)`); C are ground `holdsat` or `occurred` facts that the designer believes to be important and represents the *conditional expected output*; O are ground `holdsat` and `occurred` literals that represent the *expected output* of the use case.

The design process is iterative. A current formalisation of the model in $AnsProlog$ is tested against a set of use cases. Together with the $AnsProlog$ specification of the normative framework we add the observed events and a constraint indication that no answer set that does not satisfy O is acceptable. The latter is done by adding a constraint containing the negation of all the elements in O. If for some use cases the solver is not able to find an answer set (returns unsatisfiable), then a revision step is performed. All the use cases and the current formalisation are given as input to TAL. Possible revisions are provided to the designer who ultimately chooses which is the most appropriate. The success of the revision step depends on the state of the formalisation of the model. The set of supporting use cases can be extended as the design progresses to more accurate models.

In this paper we focus on the learning step and we show how a non-monotonic ILP system can be used to derive new rule. Refining existing rules (i.e. deleting rules or adding and delete conditions in rules) is a straightforward extension of the current framework. Though we do not discuss it in this paper, revision can be performed by extending the original rules with additional predicates that extend the search to deletion of conditions in rules and to exceptions as shown in [11].

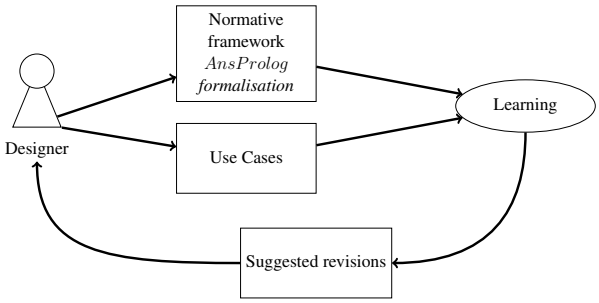

Fig. 2. Iterative design driven by use cases.

5.2 Mapping ASP to ILP

The differences between the $AnsProlog$ program and the translation into a suitable representation for TAL is procedural and only involves syntactic transformations. Thus the difference in the two representations only consists in how the inference is performed. The two semantics coincide since the same logic program is encoded and the mapping of a normative framework has exactly one answer set when given a trace. If conditions are added this can be reduced to zero.

A normative model $\mathcal{F}$ corresponds to a $AnsProlog$ program $P_{\mathcal{F}}$ as described in Section 4. All the normal clauses contained in $P_{\mathcal{F}}$ are part of B; the only differences involve time points, that are handled in B by means of a finite domain constraint solver. B also contains all the facts in C and T (negated facts are encoded by adding exceptions to the definitions of `holdsat` and `occurred`). The set of examples E contains the literals in O. Each $H \in ILP\langle E, B, S\rangle$ represents a possible revision for $\mathcal{P}$ and thus for the original normative model.

6 Example

To illustrate the capabilities of the norm learning mechanism, we have developed a relatively simple scenario that, at the same time, is complicated enough to demonstrate the key properties with little extraneous detail.

The active parties—agents—of the scenario each find themselves initially in the situation of having ownership of several (digital) objects—the blocks—that form part of some larger composite (digital) entity—a file. An agent may give a copy of one its blocks in exchange for a copy of another block with the aim of acquiring a complete set of all the blocks. For simplicity, in the situation we analyse here, we assume that initially each agent holds the only copy of a given block, and that is there is only one copy of each block in the agent population. Furthermore, we do not take into account the possibility of exchanging a block for one that the agent already has. We believe that neither of these issues does more than complicate the situation by adding more states that would obscure the essential properties that we seek to demonstrate. Thus, we arrive at a statement of the example: two agents, Alice and Bob, each holding two blocks from a set of four and each having the goal of owning all four by downloading the blocks they miss from the other while sharing, with another agent, the ones it does.

```
% Normative and Domain Rules
initiated(hasBlock(Agent, Block), I) ←
        occurred(myDownload(Agent, Block), I), holdsat(live(filesharing), I).
initiated(perm(myDownload(Agent, Block)), I) ←
        occurred(myShare(Agent), I), holdsat(live(filesharing), I).
terminated(pow(filesharing, myDownload(Agent, Block)), I) ←
        occurred(myDownload(Agent, Block), I), holdsat(live(filesharing), I).
terminated(needsBlock(Agent, Block), I) ←
        occurred(myDownload(Agent, Block), I), holdsat(live(filesharing), I).
terminated(pow(filesharing, myDownload(Agent, Block)), I) ←
        occurred(misuse(Agent), I), holdsat(live(filesharing), I).
terminated(perm(myDownload(Agent, Block)), I) ←
        occurred(myDownload(Agent, Block), I), holdsat(live(filesharing), I).
occurred(myDownload(AgentA, Block), I) ←
        occurred(download(AgentA, AgentB, Block), I), holdsat(hasBlock(AgentB, Block), I),
        holdsat(pow(filesharing, myDownload(AgentA, Block)), I), AgentA! = AgentB.
occurred(myShare(AgentB), I) ←
        occurred(download(AgentA, AgentB, Block), I), holdsat(hasBlock(AgentB, Block), I),
        holdsat(pow(filesharing, myDownload(AgentA, Block)), I), AgentA! = AgentB.
occurred(misuse(Agent), I) ← occurred(viol(myDownload(Agent, Block)), I), i).
% Initial state
holdsat(pow(filesharing, myDownload(Agent, Block)), i0).
holdsat(pow(filesharing, myShare(Agent)), i0).
holdsat(perm(download(AgentA, AgentB, Block)), i0)).
holdsat(perm(myDownload(Agent, Block)), i0).
holdsat(perm(myShare(Agent)), i0).
holdsat(hasBlock(alice, x1), i0).        holdsat(hasBlock(alice, x2), i0).
holdsat(hasBlock(bob, x3), i0).          holdsat(hasBlock(bob, x4), i0).
holdsat(needsBlock(alice, x3), i0).      holdsat(needsBlock(alice, x4), i0).
holdsat(needsBlock(bob, x1), i0).        holdsat(needsBlock(bob, x2), i0).
holdsat(live(filesharing), i0).
% fluent rules
holdsat(P, J) ← holdsat(P, I), not terminated(P, I), next(I, J).
holdsat(P, J) ← initiated(P, I), next(I, J).
occurred(E, I) ← evtype(E, ex), observed(E, I).
occurred(viol(E), I) ←
        occurred(E, I), not holdsat(perm(E), I), holdsat(live(X), I), evinst(E, X).
occurred(viol(E), I) ←
        occurred(E, I), evtype(E, inst), not holdsat(perm(E), I), event(viol(E)).
```

Fig. 3. Translation of the "sharing" normative framework into $AnsProlog$ (types omitted).

We model this as a simple normative framework, where the brute event [20] of downloading a block initiates several normative events, but the act of downloading revokes the permission of that agent to download another block until it has shared (this the complementary action to download) a block with another agent. Violation of this norm results in the download power being revoked permanently. In this way reciprocity is assured by the normative framework. Initially, each agent is empowered and permitted to share and to download, so that either agent may initiate a download operation.

Fig. 3 shows the $AnsProlog$ representation of the complete normative framework representing this scenario. In the following examples a variety of normative rules will be deliberately removed and re-learned.

6.1 Learning Setting

To show how different parts of the formal model can be learned we start from a correct specification and, after deleting some of the rules, we use TAL to reconstruct the missing parts based on a single use case. In our example TAL is set to learn hypotheses of at most three rules with at most three conditions. The choice of an upper bound on

the complexity (number of literals) of the rule ultimately rests on the final user. Alternatively, TAL can iterate on the complexity or perform a best first search that returns increasingly more complex solutions. We use the following mode declarations, M:

$m1 : modeh(terminated(perm(myDownload(+agent, +block)), +instant)).$
$m2 : modeh(initiated(perm(myDownload(+agent, +block)), +instant)).$
$m3 : modeb(occurred(myDownload(+agent, +block), +instant)).$
$m4 : modeb(occurred(myDownload(+agent, -block), +instant)).$
$m5 : modeb(occurred(myShare(+agent), +instant)).$
$m6 : modeb((+agent != +agent)).$
$m7 : modeb(holdsat(hasblock(+agent, +block), +instant)).$
$m8 : modeb(holdsat(powfilesharing(myDownload(+agent, +block)), +instant)).$

The first two mode declarations state that terminate and initiate permission rules for the normative fluent $myDownload$ can be learned. The other declarations constrain the structure of the body. The difference between $m3$ and $m4$ is that the former must refer to the same block as the one in the head of the rule while the latter introduces a possibly different block. $m8$ is an inequality constraint between agents. In general more mode declarations should be considered (e.g. initiation and termination of all types of fluents should be included) but the revision can be guided by the designer. For example new changes to a stable theory are more likely to contain errors and thus can be isolated in the revision process. The time to compute all the reported hypotheses ranges from 30 to 500 milliseconds on a 2.8 GHz Intel Core 2 Duo iMac with 2 GB of RAM.

The background knowledge B contains the rules in Fig. 3 together with the traces T given in the use cases. C in this example is empty to allow for the demonstration of the most general types of learning.

Learning a single terminate/initiate rule We suppose one of the $initiate$ rules is missing from the current specification:

$$initiated(perm(myDownload(Agent, Block)), I) \leftarrow occurred(myShare(Agent), I).$$

The designer inputs the following observed events that show how in a two agent scenario, one of the agents loses permission to download after downloading a block and reacquires it after providing a block for another agent. The trace T looks like:

$$observed(download(alice, bob, x3), 0).$$
$$observed(download(bob, alice, x1), 1).$$

The expected output O is:

$$not\ holdsat(perm(myDownload(alice, x4)), 1).$$
$$holdsat(perm(myDownload(alice, x4)), 2).$$

The trace is disfunctional if the expected output is not true in the answer set of $T \cup B$. The ILP task is thus to find a set of rules H within the language bias specified by mode declarations in M such that given the background knowledge B in Fig. 3 and the

given expected output O as conjunction of literals, O is true in the only answer set of $B \cup T \cup H$ (if one exists). TAL produces the following hypotheses:

$$initiated(perm(myDownload(A, \_)), C) \leftarrow \quad (H_1)$$
$$occurred(myShare(A), C).$$

and

$$terminated(perm(myDownload(\_, \_)), \_). \quad (H_2)$$
$$initiated(perm(myDownload(A, \_)), C) \leftarrow$$
$$occurred(myShare(A), C).$$

The second solution is not the one intended but it still supports the use case. Note that according to current implementation, whenever a fluent f is both initiated and terminated at the same time point, f still holds at the subsequent time point.

Learning multiple rules In this scenario two rules are missing from the specification:

$$initiated(perm(myDownload(Agent, Block)), I) \leftarrow$$
$$occurred(myShare(Agent), I).$$
$$terminated(perm(myDownload(Agent, Block2)), I) \leftarrow$$
$$occurred(myDownload(Agent, Block1), I).$$

We use the same T and O as previously. TAL produces the following hypotheses:

$$terminated(perm(myDownload(A, \_)), C) \leftarrow \quad (H_1)$$
$$occurred(myDownload(A, \_), C).$$
$$initiated(perm(myDownload(A, \_)), C) \leftarrow$$
$$occurred(myShare(A), C).$$
$$terminated(perm(myDownload(\_, \_)), \_). \quad (H_2)$$
$$initiated(perm(myDownload(A, \_)), C) \leftarrow$$
$$occurred(myShare(A), C).$$

The second solution is consistent with the use case, but the designer can easily discard it, since the rule is not syntactically valid with respect to the normative framework: a fluent can only be terminated as a consequence of the occurrence of an event. Using more advanced techniques for the language bias specification it would be possible to rule out such a hypothesis.

Learning of undesired violation We assume the following rule is missing:

$$initiated(perm(myDownload(Agent, Block)), I) \leftarrow$$
$$occurred(myShare(Agent), I).$$

This time we provide a different trace T:

$$observed(download(alice, bob, x3), 0).$$
$$observed(download(bob, alice, x1), 1).$$
$$observed(download(alice, bob, x4), 2).$$

As a result of the trace, a violation at time point 2 is implied that the designer knows to be undesired. The expected output is:

$$not\ occurred(viol(myDownload(alice, x4)), 2).$$

$$
\begin{array}{ll}
occurred(myShare(A), B) \leftarrow & (H1)\\
\quad occurred(download(C, A, E), B), A! = C, & \\
\quad holdsat(pow(filesharing, myDownload(A, E)), B). & \\
occurred(myShare(A), B) \leftarrow & (H2)\\
\quad occurred(download(C, A, E), B), A! = C, & \\
\quad holdsat(pow(filesharing, myDownload(A, E)), B), & \\
\quad holdsat(hasblock(A, E), B). & \\
occurred(myShare(A), B) \leftarrow & (H3)\\
\quad occurred(download(C, A, E), B), A! = C, & \\
\quad holdsat(pow(filesharing, myDownload(C, E)), B). & \\
occurred(myShare(A), B) \leftarrow & (H4)\\
\quad occurred(download(C, A, E), B), A! = C, & \\
\quad holdsat(pow(filesharing, myDownload(C, E)), B), & \\
\quad holdsat(hasblock(A, E), B). & \\
occurred(myShare(A), B) \leftarrow & (H5)\\
\quad occurred(download(C, A, E), B), A! = C, & \\
\quad holdsat(hasblock(A, E), B). & \\
occurred(myShare(A), B) \leftarrow & (H6)\\
\quad occurred(download(C, A, E), B), & \\
\quad holdsat(pow(filesharing, myDownload(C, E)), B). &
\end{array}
$$

Fig. 4. Proposals to revise the generate rule

The outcome of the learning consists of the following two possible solutions:

$$
\begin{array}{ll}
initiated(perm(myDownload(A, \_)), C) \leftarrow & (H_1)\\
\quad occurred(myShare(A), C). & \\
initiated(perm(myDownload(\_, \_)), \_). & (H_2)
\end{array}
$$

that show how the missing rule is derived from the undesired violation. As in the previous scenario the designer can easily dismiss the second candidate.

Learning a generate rule To account for the different type of rules that need to be learned, the language bias is extended to consider learning of generate rules. The new mode declarations are:

$$
\begin{array}{l}
modeh(occurred(myShare(+agent), +instant)).\\
modeb(occurred(download(-agent, +agent, -block), +instant)).
\end{array}
$$

We use the same trace and expected output as in the previous scenario (three observed events). The following rule is eliminated from the specification:

$$
\begin{array}{l}
occurred(myShare(AgentB), I) \leftarrow\\
\quad AgentA! = AgentB,\\
\quad occurred(download(AgentA, AgentB, Block), I),\\
\quad holdsat(hasblock(AgentB, Block), I),\\
\quad holdsat(pow(filesharing, myDownload(AgentA, Block)), I).
\end{array}
$$

This is the most complicated case for the designer as a set of six different hypotheses are returned by TAL (see Figure 4). Knowing the semantics of the function symbol `download(AgentA, AgentB, Block)` as `AgentA` downloads from `AgentB` the designer should be able to select the most appropriate rule.

7 Related Work

The motivation behind this paper is the problem of how to converge upon a complete and correct normative framework *with respect to the intended range of application*, where in practice these properties may be manifested by incorrect or unexpected behaviour in use. Additionally, we would observe, from practical experience with our particular framework, that it is often desirable, as with much software development, to be able to develop and test incrementally—and regressively—rather than attempt verification once the system is (notionally) complete.

The literature seems to fall broadly into three categories: (a) concrete language frameworks (OMASE, Operetta, InstSuite, MOISE, Islander, OCeAN and the constraint approach of Garcia-Camino (full references to these are currently omitted because of page limitations)) for the specification of normative systems, that are typically supported by some form of model-checking, and in some cases allow for change in the normative structure; (b) logical formalisms, such as [16], that capture consistency and completeness via modalities and other formalisms like [5], that capture the concept of norm change, or [**?**] and [**?**]; (c) mechanisms that look out for (new) conventions and handle their assimilation into the normative framework over time and subject to the current normative state and the position of other agents [2, 8]. Essentially, the objective of each of the above is to realize a transformation of the normative framework to accommodate some form of shortcoming. These shortcomings can be identified in several ways: (a) by observing that a particular state is rarely achieved, which can indicate there is insufficient normative guidance for participants, or (b) a norm conflict occurs, such that an agent is unable to act consistently under the governing norms [23], or (c) a particular violation occurs frequently, which may indicate that the violation conflicts with an effective course of action that agents prefer to take, the penalty notwithstanding. All of these can be viewed as characterising emergent [28] approaches to the evolution of normative frameworks, where some mechanism, either in the framework, or in the environment, is used to revise the norms. In the approach taken here, the designer presents use cases that effectively capture their behavioural requirements for the system, in order to 'fix' bad states. This has an interesting parallel with the scheme put forward by Serrano and Saugar [30], where they propose the specification of incomplete theories and their management through incomplete normative states identified as "pending". The framework lets designated agents resolve this category through the speech acts *allow* and *forbid* and scheme is formalised using an action language.

A useful categorisation of normative frameworks appears in [6]. Whether the norms here are 'strong' or 'weak' —the first guideline— depends on whether the purpose of the normative model is to develop the system specification or additionally to provide an explicit representation for run-time reference. Likewise, in respect of the remaining guidelines, it all depends on how the framework we have developed is actually used: we have chosen, for the purpose of this presentation, to stage norm refinement so that it is an off-line (in the sense of prior to deployment) process, while much of the discussion in [6] addresses run-time issues. Whether the process we have outlined here could effectively be a means for on-line mechanism design, is something we have yet to explore.

From an ILP perspective, we employ an ILP system that can learn logic programs with negation (stratified or otherwise). Though recently introduced and in its early stages of development *TAL* is the most appropriate choice to support this work for two main reasons: it is supported by completeness results, unlike other existing non-monotonic ILP systems ([27], [22]), and it can be tailored to particular requirements (e.g. different search strategies can address performance requirements). The approach presented in this paper is related to other recently proposed frameworks for the elaboration of formal specifications via inductive learning. Within the context of software engineering, [**?**] has shown how examples of desirable and undesirable behaviour of a software system can be used by an ILP system, together with an incomplete background knowledge of the envisioned system and its environment, to compute missing requirements specifications. A more general framework has been proposed [**?**] where desirable and undesirable behaviours are generated from counterexamples produced by model checking a given (incomplete) requirements specification with respect to given system properties. The learning of missing requirements has in this case the effect of eliminating the counterexamples by elaborating further the specification.

8 Conclusions and Future Work

We have presented an approach for learning norms and behavioural rules, via inductive logic programming, from example traces in order to guide and support the synthesis of a normative framework. This addresses a crucial problem in normative systems as the development of such specifications is in general a manual and error-prone task. The approach deploys an established inductive logic programming system [12] that takes in input an initial (partial) description of a normative system and use cases of expected behaviours provided by the designer and generates hypothesis in the form of missing norms and behavioural rules that together with the given description explain the use cases. Although the approach presented in this paper has been tailored for learning missing information, it can also be applied to computing revisions over the existing description. In principle this can be achieved by transforming the existing normative rules into defeasible rules with exceptions and using the same ILP system to compute exception rules. These exceptions would in essence be prescriptions for changes (i.e. addition and/or deletion of literals in the body of existing rules) in the current specification. An appropriate refactoring of the defeasible rules based on the learned exception rules would give a revised (non-defeasible) specification. In this case, the revision would be in terms of changes over the rules of a normative framework instead of changes over its belief state, as would be the case if a TMS approach were adopted.

There are several criticisms that can be levelled at the approach as it stands. Firstly, the design language is somewhat unfriendly: a proper tool would have a problem-oriented language, like InstAL/QL [10, 19]. A system designer would then start from an initial description of their normative framework with some use cases and receive automated suggestions of additional norms to include in the framework written in the same high-level language. The machinery described here, based on ASP syntax and ILP formulation, would then be used as a sound “back- end” computation to a formalism familiar to the system designer. Secondly, better control is needed over the rules that are

learned and over the filtering of incorrect rules; at present this depends on specialised knowledge of the learning process. This can to some extent be controlled through careful choice of and limits on the size of use cases—probably involving heuristics—to improve the effectiveness of the learning process in the search for relevant hypotheses and pruning of those potential solutions that cannot be translated back into the canonical form of the normative framework. Despite these issues, we believe we have identified an interesting path for automating and development and debugging of practical normative specifications and perhaps, in the long term, a mechanism for on-line norm evolution.

References

1. A. Artikis, M. Sergot, and J. Pitt. Specifying electronic societies with the Causal Calculator. In *Proceedings of workshop on agent-oriented software engineering iii (aose)*, LNCS 2585. Springer, 2003.
2. Alexander Artikis. Dynamic protocols for open agent systems. In Sierra et al. [31], pages 97–104.
3. Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge Press, 2003.
4. Guido Boella, Pablo Noriega, Gabriella Pigozzi, and Harko Verhagen, editors. *Normative Mult-Agent Systems*, number 09121 in Dagstuhl Seminar Proceedings. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2009.
5. Guido Boella, Gabriella Pigozzi, and Leendert van der Torre. Normative framework for normative system change. In Sierra et al. [31], pages 169–176.
6. Guido Boella, Gabriella Pigozzi, and Leendert van der Torre. Normative systems in computer science - ten guidelines for normative multiagent systems. In *Normative Mult-Agent Systems*, 2009.
7. Guido Boella and Leendert van der Torre. Constitutive Norms in the Design of Normative Multiagent Systems, City College. In *Proceedings of the Sixth International Workship on Computational Logic in Multi-Agent Systems (CLIMA-VI)*, June 2005.
8. George Christelis and Michael Rovatsos. Automated norm synthesis in an agent-based planning environment. In Sierra et al. [31], pages 161–168.
9. Owen Cliffe, Marina De Vos, and Julian Padget. Answer set programming for representing and reasoning about virtual institutions. In *Computational Logic for Multi-Agents (CLIMA VII)*, volume 4371 of *LNAI*, pages 60–79. Springer, May 2006.
10. Owen Cliffe, Marina De Vos, and Julian A. Padget. Embedding landmarks and scenes in a computational model of institutions. In *Coordination, Organizations, Institutions, and Norms in Agent Systems III*, volume 4870 of *LNCS*, pages 41–57, September 2008.
11. D. Corapi, O. Ray, A. Russo, A.K. Bandara, and E.C. Lupu. Learning rules from user behaviour. In *5th Aritificial Intelligence Applications and Innovations (AIAI 2009)*, April 2009.
12. D. Corapi, A. Russo, and E. Lupu. Inductive logic programming as abductive search. In *26th International Conference on Logic Programming, Leibniz International Proceedings in Informatics*. Schloss Dagstuhl Research Online Publication Server, 2010.
13. Marc Denecker and Danny De Schreye. Sldnfa: An abductive procedure for abductive logic programs. *J. Log. Program.*, 34(2):111–167, 1998.
14. V. Dignum. *A model for organizational interaction: based on agents, founded in logic*. PhD thesis, University of Utrecht, 2004.
15. Sašo Džroski. Relational data mining applications: an overview. pages 339–360, 2000.
16. Christophe Garion, Stéphanie Roussel, and Laurence Cholvy. A modal logic for reasoning on consistency and completeness of regulations. In Boella et al. [4].

17. M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-Driven Answer Set Solving. In *Proceeding of IJCAI07*, pages 386–392, 2007.
18. Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3-4):365–386, 1991.
19. Luke Hopton, Owen Cliffe, Marina De Vos, and Julian Padget. Instql: A query language for virtual institutions using answer set programming. In *Proceedings of the 10th International Workshop on Computational Logic in Multi-Agent Systems (ClimaX)*, IfI Technical Report Series, pages 87–104, September 2009.
20. John R. Searle. *The Construction of Social Reality*. Allen Lane, The Penguin Press, 1995.
21. Andrew J.I. Jones and Marek Sergot. A Formal Characterisation of Institutionalised Power. *ACM Computing Surveys*, 28(4es):121, 1996. Read 28/11/2004.
22. Tim Kimber, Krysia Broda, and Alessandra Russo. Induction on failure: Learning connected horn theories. In *LPNMR*, pages 169–181, 2009.
23. Martin Kollingbaum, Timothy Norman, Alun Preece, and Derek Sleeman. Norm conflicts and inconsistencies in virtual organisations. In *Proceedings of COIN 2006*, volume 4386 of *LNCS*, pages 245–258. Springer, 2007.
24. N. Lavrač and S. Džeroski. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, 1994.
25. I. Niemelä and P. Simons. Smodels: An implementation of the stable model and well-founded semantics for normal LP. In *LPNMR*, volume 1265 of *LNAI*, pages 420–429. Springer, July 28–31 1997.
26. Juan A. Rodriguez-Aguilar. *On the Design and Construction of Agent-mediated Institutions*. PhD thesis, Universitat Autonoma de Barcelona, 2001.
27. Chiaki Sakama. Nonmonotonic inductive logic programming. In *LPNMR*, page 62, 2001.
28. Bastin Tony Roy Savarimuthu and Stephen Cranefield. A categorization of simulation works on norms. In Boella et al. [4].
29. Marek Sergot. (C+)++: An Action Language For Representing Norms and Institutions. Technical report, Imperial College, London, August 2004.
30. Juan-Manuel Serrano and Sergio Saugar. Dealing with incomplete normative states. In *Proceedings of COIN 2009*, volume 6069 of *LNCS*. Springer, 2010. in press.
31. Carles Sierra, Cristiano Castelfranchi, Keith S. Decker, and Jaime Simão Sichman, editors. *8th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009), Budapest, Hungary, May 10-15, 2009, Volume 1*. IFAAMAS, 2009.
32. Munindar P. Singh. A social semantics for agent communication languages. In *Issues in Agent Communication*, pages 31–45. Springer-Verlag: Heidelberg, Germany, 2000.