TLP **11** (4–5): 783–799, 2011.
doi:10.1017/S1471068411000305

Normative design using inductive learning

DOMENICO CORAPI and ALESSANDRA RUSSO

Department of Computing, Imperial College London, 180 Queen's Gate, SW7 2AZ, London, UK
(*e-mail:* {d.corapi, a.russo}@ic.ac.uk)

MARINA DE VOS and JULIAN PADGET

Department of Computing, University of Bath, BA2 7AY, Bath, UK
(*e-mail:* {mdv,jap}@cs.bath.ac.uk)

KEN SATOH

Principles of Informatics Research Division, National Institute of Informatics, Chiyoda-ku, 2-1-2, Hitotsubashi, Tokyo 101-8430, Japan
(*e-mail:* ksatoh@nii.ac.jp)

Abstract

In this paper we propose a use-case-driven iterative design methodology for normative frameworks, also called virtual institutions, which are used to govern open systems. Our computational model represents the normative framework as a logic program under answer set semantics (ASP). By means of an inductive logic programming approach, implemented using ASP, it is possible to synthesise new rules and revise the existing ones. The learning mechanism is guided by the designer who describes the desired properties of the framework through use cases, comprising (i) event traces that capture possible scenarios, and (ii) a state that describes the desired outcome. The learning process then proposes additional rules, or changes to current rules, to satisfy the constraints expressed in the use cases. Thus, the contribution of this paper is a process for the elaboration and revision of a normative framework by means of a semi-automatic and iterative process driven from specifications of (un)desirable behaviour. The process integrates a novel and general methodology for theory revision based on ASP.

KEYWORDS: normative frameworks, inductive logic programming, theory revision

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. 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. Normative frameworks allow 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.

Just as legislators, and societies, find inconsistencies in their rules (or conventions), so too may designers of normative frameworks. The details of the specification makes it relatively easy to miss crucial operations needed to help or inhibit intended behaviour. In order to make an analogy with software engineering, this characterises the gap between requirements and implementation and what we describe here can be seen as an automated mechanism to support the validation of normative frameworks, coupled with regression testing.

The contribution of the work is two-fold. Firstly, we show how inductive logic programming (ILP) can be used to fill gaps in the rules of an existing normative framework. The designer normally develops a system with a certain behaviour in mind. This intended behaviour can be captured in *use cases*, which comprise two components: (a) a description of a scenario, and (b) the expected outcome when executing the scenario. Use cases are added to the program to validate the existence of an answer set. Failure to solve the program indicates that the specification does not yield the intended behaviour. In this case, the program and the failing use case(s) are given to an inductive learning tool, which will then return suggestions for improving the normative specification such that the use cases are satisfied. Secondly, we present a novel integrated methodology for theory revision that can be used to revise a logic program under the answer set semantics/programming (ASP) and supports the development process by associating answer sets (that can be used for debugging purposes) to proposed revisions. Due to the non-monotonic nature of ASP, the designer can provide the essential parts of the use case creating a template rather that a fully specified description. The revision mechanism is general and can be applied to other domains. We demonstrate the methodology through a case study showing the iterative revision process.

The paper is organised as follows. Section 2 presents some background material on the normative framework, while Section 3 introduces the ILP setting used in our proposed approach. Section 4 illustrates the methodology and how the revision task can be formulated into an ILP problem. We illustrate the flexibility and expressiveness of our approach through specifications of a reciprocal file sharing normative system. Section 5 discusses the details of the revision mechanism and the learning system. Section 6 relates our approach to existing work. We conclude with a summary and remarks on future work.

2 Normative frameworks

The essential idea of normative frameworks is a (consistent) collection of rules whose purpose is to describe ‘a principle of right action binding upon the members of a group and serving to guide, control, or regulate proper and acceptable behaviour’ (*Merriam-Webster Dictionary*). These rules may be stated in terms of events, specifically the events that matter for the functioning of the normative framework.

2.1 Formal model

The formalisation of the above may be defined as conditional operations on a set of terms that represents the normative state. In order to provide the context for this paper, we give an outline of a formal event-based model for the specification of normative frameworks

https://doi.org/10.1017/S1471068411000305 Published online by Cambridge University Press

$\mathcal{N} = \langle \mathcal{E}, \mathcal{F}, \mathcal{C}, \mathcal{G}, \mathcal{I} \rangle$, where

1. $\mathcal{F} = \mathcal{W} \cup \mathcal{P} \cup \mathcal{O} \cup \mathcal{D}$
2. $\mathcal{G} : \mathcal{X} \times \mathcal{E} \rightarrow 2^{\mathcal{E}_{norm}}$
3. $\mathcal{C} : \mathcal{X} \times \mathcal{E} \rightarrow 2^{\mathcal{F}} \times 2^{\mathcal{F}}$ where $\mathrm{C}(X, \mathtt{e}) =$ $(\mathcal{C}^{\uparrow}(\phi, \mathtt{e}), \mathcal{C}^{\downarrow}(\phi, \mathtt{e}))$ where
   (i) $\mathcal{C}^{\uparrow}(\phi, \mathtt{e})$ initiates a fluent
   (ii) $\mathcal{C}^{\downarrow}(\phi, \mathtt{e})$ terminates a fluent
4. $\mathcal{E} = \mathcal{E}_{ex} \cup \mathcal{E}_{norm}$ with $\mathcal{E}_{norm} = \mathcal{E}_{act} \cup \mathcal{E}_{viol}$
5. $\mathcal{I}$
6. State Formula: $\mathcal{X} = 2^{\mathcal{F} \cup \neg \mathcal{F}}$

(a)

$$p \in \mathcal{F} \Leftrightarrow \mathtt{ifluent(p)}. \quad (1)$$
$$e \in \mathcal{E} \Leftrightarrow \mathtt{event(e)}. \quad (2)$$
$$e \in \mathcal{E}_{ex} \Leftrightarrow \mathtt{evtype(e, obs)}. \quad (3)$$
$$e \in \mathcal{E}_{act} \Leftrightarrow \mathtt{evtype(e, act)}. \quad (4)$$
$$e \in \mathcal{E}_{viol} \Leftrightarrow \mathtt{evtype(e, viol)}. \quad (5)$$
$$\mathcal{C}^{\uparrow}(\phi, \mathtt{e}) = P \Leftrightarrow \forall p \in P \cdot \mathtt{initiated(p, T)} \leftarrow \text{occurred}(\mathtt{e}, \mathtt{I}), EX(\phi, T). \quad (6)$$
$$\mathcal{C}^{\downarrow}(\phi, \mathtt{e}) = P \Leftrightarrow \forall p \in P \cdot \mathtt{terminated(p, T)} \leftarrow \text{occurred}(\mathtt{e}, \mathtt{I}), EX(\phi, T). \quad (7)$$
$$\mathcal{G}(\phi, \mathtt{e}) = E \Leftrightarrow g \in E, \mathtt{occurred(g, T)} \leftarrow \mathtt{occurred(e, T)}, \mathtt{holdsat(pow(e), I)}, EX(\phi, \mathrm{T}). \quad (8)$$
$$p \in \mathcal{I} \Leftrightarrow \mathtt{holdsat(p, i00)}. \quad (9)$$

(b)

Fig. 1. (a) Formal specification of the normative framework, and (b) translation of normative framework-specific rules into *AnsProlog* .

that captures all the essential properties, namely *empowerment*, *permission*, *obligation* and *violation*. We adopt the formalisation from Cliffe *et al*. (2006), summarized in Figure 1(a), because of its straightforward mapping to ASP.

The essential elements of the normative framework are events ($\mathscr{E}$), which bring about changes in state, and fluents ($\mathscr{F}$), which 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 types of events: normative events ($\mathscr{E}_{norm}$) that are the events defined by the framework, and exogenous events ($\mathscr{E}_{ex}$), some of whose occurrence may trigger normative events in a direct reflection of 'counts-as' (Jones and Sergot 1996), and others that are of no relevance to this particular framework. Normative events are further partitioned into normative actions ($\mathscr{E}_{act}$) that denote changes in normative state, and violation events ($\mathscr{E}_{viol}$) that signal the occurrence of violations. Violations may arise either from explicit generation (i.e. from the occurrence of a non-permitted event), or from the non-fulfilment of an obligation. We also distinguish two types of fluents: *normative fluents* that denote normative properties of the state such as *permissions* ($\mathscr{P}$), *powers* ($\mathscr{W}$) and obligations ($\mathscr{O}$), and *domain fluents* ($\mathscr{D}$) that correspond to properties specific to a particular normative framework. A normative state is represented by the fluents that hold true in this state. Fluents that are not present are considered to be false. Conditions on a state ($\mathscr{X}$) are expressed by a set of fluents that should be true or false. When the creation event occurs, the normative state is initialised with the fluents specified in $\mathscr{I}$.

Changes in a normative state are achieved through the definition of two relations: (i) the generation relation ($\mathscr{G}$) that 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 ($\mathscr{C}$) that specifies the initiation and termination of fluents, subject to the performance of some action in a state matching some condition.

The semantics of a 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: ind (i) the transitive closure of $\mathscr{G}$ with respect to a given exogenous

https://doi.org/10.1017/S1471068411000305 Published online by Cambridge University Press

event determines all the generated (normative) events, iind (ii) to this all violations of non-permitted events and non-fulfilled obligations are added, giving the set of all events whose consequences determine the new state iiind (iii) the application of $\mathscr{C}$ to this set of events identifies all fluents that are initiated and terminated with respect to the current state, so determining 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 ASP (see Section 2.2) and it is this representation that is used in the learning process described in Section 4. A detailed example of formal model of an institution can be found in Cliffe *et al*. (2006).

2.2 Computational model

The formal model described above can be translated into an equivalent computational model using ASP (Gelfond and Lifschitz 1991) with *AnsProlog* as an implementation language. *AnsProlog* is a knowledge representation language that allows 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. For our mapping we followed the naming convention used in the event calculus (Kowalski and Sergot 1986) and action languages (Gelfond and Lifschitz 1998).

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*. *Literals* are atoms a or negated atoms not a. We say that not a is true if we cannot find evidence supporting the truth of a. Atoms and literals are used to create rules of the general form: $a \leftarrow b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n$, where a, b_i and c_j are atoms. Intuitively, this means *if all atoms b_i are known/true and no atom c_j is known/true, then a must be known/true*. We refer to a as the head and $b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n$ as the body of the rule. Rules with empty body are called *facts*. Rules with empty head are referred to as *constraints*, indicating that no solution should be able to satisfy the body. A *(normal) program (or theory)* is a conjunction of rules and is also denoted by a set of rules. The semantics of *AnsProlog* is 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 may have zero or more answer sets, each corresponding to a solution.

The mapping of a normative framework consists of three parts: a *base component* that is independent of the framework being modelled, the *time component* and the *framework-specific component*. The independent component deals with inertia of the fluents, the generation of violation events of non-permitted actions and of unfulfilled obligations. The time component defines the predicates for time and is responsible for generating a single observed event at every time instance. The mapping uses the following atoms: `ifluent(p)` to identify fluents, `evtype(e,t)` to describe the type of an event, `event(e)` to denote the events, `instant(i)` for time instances, `final(i)` for the last time instance, `next(i1,i2)` to establish time ordering, `occurred(e,i)` to indicate that the (normative) event happened at time i, `observed(e,i)` that the (exogenous) event was observed at time i, `holdsat(p,i)` to state that the normative fluent p holds at i and finally `initiated(p,i)` and `terminated(p,i)` for fluents that are initiated and terminated at i. Note that exogenous events are always empowered so that observed events are always occurred events, but

https://doi.org/10.1017/S1471068411000305 Published online by Cambridge University Press

that normative events are not, so their occurrence is conditional on their empowerment. Figure 1(b) provides the framework-specific translation rules, including the definition of all fluents and events as facts. We translate expressions into *AnsProlog* rule bodies as conjunctions of literals using negation as failure for negated expressions.

The translation of the formal model is augmented with a trace program, specifying the length of traces that the designer is interested in and rules to ensure that all but the final time instance is associated with exactly one exogenous event. Specific occurrences of events can be specified as facts (e.g. `observed(event,instance)`). We refer to a complete trace when all exogenous events for a given time interval are specified. If a trace is incomplete when the model needs to determine the missing exogenous events. While not discussed in this paper, both normative framework and learning tool can deal with both types of traces. When the model is supplemented with the *AnsProlog* specification of a complete trace, we obtain a single answer set corresponding to the model matching the trace.[1] In this case the complexity of computing the answer set is linear with respect to the number of time instance being modelled. This result can be easily derived from the structure of the program. Of course, in the absence of a complete trace, the complexity is NP-complete, as the traces composed of all possible combinations of missing exogenous events are computed. See Cliffe (2007) for further details and proofs.

3 Learning

Inductive Logic Programming (Muggleton 1995) is a machine learning technique concerned with the induction of logic theories that generalise (positive and negative) examples with respect to a prior background knowledge. For example, from the observations (properties in this paper) $P_{fly} = \{fly(a), fly(b), \text{not } fly(c)\}$ and a background knowledge containing the two facts $bird(a)$ and $bird(b)$, we can generalise the concept $fly(X) \leftarrow bird(X)$. In non-trivial problems it is crucial to define the space of possible solutions accurately. Target theories are within a space defined by a *language bias* that can be expressed using the notion of mode declaration (Muggleton 1995).

Definition 1
A *mode declaration* is either a *head declaration*, written *modeh*(s), or a *body declaration*, written *modeb*(s), where s is a *schema*. A schema is a ground literal containing special terms called *placemarkers*. A *placemarker* is either '$+type$', '$-type$' or '$\#type$', where *type* denotes the type of the placemarker and the three symbols '+', '−' and '#' indicate that the placemarker is an input, an output and a constant, respectively.

In the previous example a possible language bias would be expressed by three mode declarations in M_{fly}: $modeh(fly(+animal))$, $modeb(bird(+animal))$ and $modeb(penguin(+animal))$.

A rule $h \leftarrow b_1, \ldots, b_n$ is *compatible* with a set M of mode declarations iff (a) h is the schema of a head declaration in M and b_i are the schemas of body declarations in M, where every input and output placemarkers are replaced by variables, and constant placemarkers

[1] The structure of the program (the stratified base part and observed events as facts) guarantees that the program has exactly one answer set. See Cliffe (2007) for further details and proofs.

https://doi.org/10.1017/S1471068411000305 Published online by Cambridge University Press

are replaced by constants; (b) every input variable in any atom b_i is either an input variable in h or an output variable in some $b_j, j < i$ and (c) all variables and constants are of the corresponding type (enforced by implicit conditions in the body of the rules). 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. $s(M)$ is a set of all the rules compatible with M.

Definition 2
An *ILP task* is a tuple $\langle P, B, M\rangle$, where P is a set of conjunctions of literals, called *properties*, B is a normal program, called *background theory*, and M is a set of mode declarations. A theory H, called *hypothesis*, is an inductive solution for the task $\langle P, B, M\rangle$ if (i) $H \subseteq s(M)$, and (ii) P is true in all the answer sets of $B \cup H$.

Our approach for incremental development of a normative system supports the synthesis of new rules and the revision of existing one from given use-cases. We are therefore interested in the task of Theory Revision (TR). As discussed in Corapi *et al*. (2009), non-monotonic ILP can be used to revise an existing theory. The key notion is that of *minimal revision*. In general, a TR system is biased towards the computation of theories that are similar to a given revisable theory. Our revision algorithm uses a measure of minimality similar to that proposed by Wogulis and Pazzani (1993), and defined in terms of *numbers of revision operations* required to transform one theory into another.

Definition 3
Let T' and T be normal logic programs. A revision transformation r is such that $r(T) = T'$, and T' is obtained from T by deleting a rule, adding a fact, adding a condition to a rule in T or deleting a condition from a rule in T. T' is a revision of T with distance $c(T, T') = n$ iff $T' = r^n(T)$ and there is no $m < n$ such that $T' = r^m(T)$.

For example, given the theory $T_{fly} = \{fly(X) \leftarrow bird(X)\}$, $T'_{fly} = \{fly(X) \leftarrow bird(X), not\ penguin(X)\}$ is a revision of T with distance 1. Note that, although we refer to Definition 3, it is also possible to weight revisions differently or introduce different transformations.

Definition 4
A *TR* task is a tuple $\langle P, B, T, M\rangle$, where P is a set of conjunctions of literals, called *properties*, B is a normal program, called *background theory*, $T \subseteq s(M)$ is a normal program, called *revisable theory*, and M is a set of mode declarations. The theory T', called *revised theory*, is a *TR solution* for the task $\langle P, B, T, M\rangle$ with distance $c(T, T')$, iff (i) $T' \subseteq s(M)$, (ii) P is true in all the answer sets of $B \cup T'$, (iii) if a theory S exists that satisfies conditions (i) and (ii), then $c(T, S) \geqslant c(T, T')$ (i.e. minimal revision).

For example, let $B_{fly} = \{animal(X).\ bird(X).\ penguin(c).\}$, T_{fly}, P_{fly} and M_{fly} as in the previous examples. T'_{fly} is a TR solution for the task $\langle P_{fly}, B_{fly}, T_{fly}, M_{fly}\rangle$ with distance 1. The main difference with the ILP task given in Definition 2 is the availability of an initial revisable theory and the consequent bias as discussed in more detail in the following sections.

https://doi.org/10.1017/S1471068411000305 Published online by Cambridge University Press

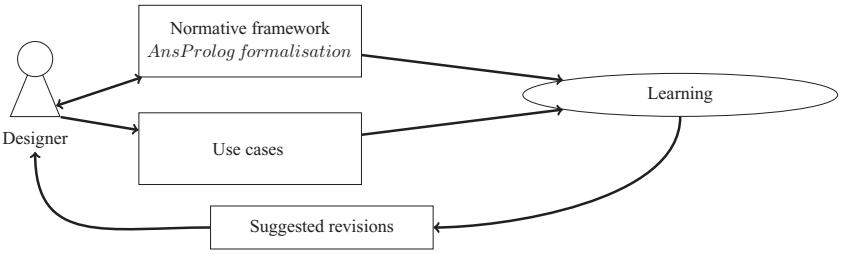

Fig. 2. Iterative design driven by use cases.

4 Revising normative rules

4.1 Methodology

Use cases represent instances of executions that are known to the designer and drive the elaboration of a normative system. If the current formalisation of a normative system does not match the intended behaviour in the use cases, then the formalisation is not complete or is incorrect, and an extension or revision is required.

Each use case $u \in U$ is a tuple $\langle T, O \rangle$, where T, a *trace*, specifies a set of exogenous events (`observed(e,t)`), and O is a set of `holdsat` and `occurred` literals that represents the *expected output* of the use case. Given a set U of use cases, T_U and O_U denote, respectively, the set of all the traces and expected outputs in all the use cases in U. The time points of different use cases relate to different instances of executions of the normative system to avoid the effect of events in one use case affecting the fluents of another use case. The use cases can, but do *not* have to, be complete traces (i.e. an event for each time instance) and expected output can contain positive as well as negative literals.

For a given translation of a normative framework N, the designer must specify what part of the theory is subject to revision. The theory is split into two parts: a 'revisable' part, N_T, and a 'fixed' part, N_B. By default the former includes rules of the form (6), (7) and (8), given in Figure 1(b), and the latter includes the rest of the representation of the normative system and the set T_U of traces in U.

Given a set U of use cases, a TR task for a normative framework $\mathcal{N}$ is defined as the tuple $\langle O_U, N_B \cup T_U, N_T, M \rangle$, where M includes by default a body declaration for any static relation declared in N_B, and the following mode declarations (where the schema is opportunely formed by substituting arguments with input placemarkers): $modeh(occurred(e^*, +instant))$, for each $e \in \mathscr{E}_{norm}$; $modeh(initiated(f^*, +instant))$ and $modeh(terminated(f^*, +instant))$, for each $f \in \mathscr{F}$; $modeb(holdsat(f^*, +instant))$, for each $f \in \mathscr{F}$; $modeb(occurred(e^*, +instant))$, for each $e \in \mathscr{E}$.

The choice of the set of mode declaration M is crucial and is ultimately the responsibility of the designer. Many mode declarations ensure higher coverage of the specification but increase the computation time. Conversely, fewer mode declarations improve performance but may result in partial solutions. The choice may be driven, for example, by previous design cycles, or interest in more problematic parts of the specification.

As shown in Figure 2, the design of a normative system is an iterative process. The representation N in *AnsProlog* of a system described by the designer using a normative language is tested against a set of use cases also provided by the designer. This analysis

https://doi.org/10.1017/S1471068411000305 Published online by Cambridge University Press

step is performed by running an ASP solver over N, extended with the observed events included in the use cases and a constraint indicating that no answer set that does not satisfy O is acceptable. Conceptually, if the solver is not able to find an answer set (i.e. returns unsatisfiable), then some of the given use cases are not satisfied in the answer sets of N and a revision step is performed. Possible revisions are provided to the designer who ultimately chooses the most appropriate one.

4.2 Case study

We illustrate the methodology with a small but rich enough case study that demonstrates the key properties and benefits of our proposed approach. The following is a description of a reciprocal file sharing normative framework.

The active parties – agents – of the scenario 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 is required to share a copy of a block they hold before they can download a copy of block that they are missing. Initially each agent holds the only copy of a given block and there is only one copy of each block in the agent population. Some *vip* agents are able to download blocks without any restriction. Agents that request a download and have not shared a block after a previous download generate a violation for the download action and a misuse violation for the agent. A misuse terminates the empowerment of the agent to download blocks.

The designer devises the following use case $\langle T, O \rangle$:

$$T = \begin{cases} observed(start, i00). \\ observed(download(alice, bob, x3), i01). \\ observed(download(charlie, bob, x3), i02). \\ observed(download(bob, alice, x1), i03). \\ observed(download(charlie, alice, x1), i04). \\ observed(download(alice, charlie, x5), i05). \\ observed(download(alice, bob, x4), i06). \end{cases} \qquad O = \begin{cases} not\ viol(myDownload(alice, x3), i01). \\ not\ viol(myDownload(charlie, x3), i02). \\ not\ viol(myDownload(bob, x1), i03). \\ not\ viol(myDownload(charlie, x1), i04). \\ not\ viol(myDownload(alice, x5), i05). \\ viol(myDownload(alice, x4), i06). \end{cases}$$

The use case models a sequence of events that includes a violation at the time point $i06$, while the *download* events at the other time points do not generate violations. In the trace, *charlie* performs a download at time point $i04$ without sharing a block after the last download. This is not expected to generate a violation, as *charlie* is defined as *vip* ($isVIP(charlie) \in N$).

The initial normative system includes the domain component and type definitions given in Figure 1(b) and a specific component given by the following revisable theory N_T:

```
%rule 1
initiated(hasblock(X,B),I) :-
        occurred(myDownload(X,B),I).
%rule 2
initiated(perm(myDownload(X,B)),I) :-
        occurred(myShare(X),I).
%rule 3
terminated(pow(extendedfilesharing,myDownload(X,B)),I) :-
        occurred(misuse(X),I).
%rule 4
terminated(perm(myDownload(X,B2)),I) :-
        occurred(myDownload(X,B),I).
%rule 5
occurred(myDownload(X,B),I) :-
        occurred(download(Y,Y,B),I), holdsat(hasblock(Y,B),I).
%rule 6
occurred(myShare(X),I) :-
        occurred(download(Y,X,B),I), holdsat(hasblock(X,B),I).
```

https://doi.org/10.1017/S1471068411000305 Published online by Cambridge University Press

Given the use case and the above formalisation of the normative system, the first iteration of our approach proposes, through the revision process, the deletion of a condition in *rule 5* and addition of a condition to *rule 4* as shown below (leaving the other rules unaltered):

```
%rule 4 - revised
terminated(perm(myDownload(X,B2)),I) :-
        not isVIP(X), occurred(myDownload(X,B),I).
%rule 5 - revised
occurred(myDownload(X,B),I) :-
        holdsat(hasblock(Y,B),I).
```

However, this is not yet the intended formalisation. As an additional debugging facility the designer can request the set of violations that is true in the answer sets that correspond to the revision and notice that unwanted violations are generated at each time point. This feedback can be used to refine the use case provided. In fact, the use case specifies the single specific violations that must *not* occur but it does not request explicitly that no violations should occur in the first five time points (e.g. *viol(myDownload(alice,x3),i02), viol(myDownload(alice,x4),i02)*). These violations can be observed in the answer set associated with the revision. The designer can then improve the use case by modifying the set of expected outputs:

$$O = \begin{cases} viol(myDownload(alice, x4), i06). \\ not\ viol(myDownload(A, B), T), T\,!= i06. \\ occurred(misuse(alice), i06). \\ not\ occurred(misuse(X), T), T\,!= i06. \end{cases}$$

In the subsequent iteration, the revision process suggests changes that include those identified in the previous iteration (i.e. addition of condition in *rule 4* and deletion of condition in *rule 5*), and the addition of a further condition in the body of *rule 5*. The combined effect of these changes fixes the original error in the specification by also changing the name of one of the variables. Furthermore, as the output O of the use case includes a desired *misuse* event, which is not currently formalised in the system, the revision also suggests the new *rule 7* given below. The final theory N'_T includes the following rules (leaving untouched rules 1, 2, 3 and 6)[2]:

```
%rule 4 - revised
terminated(perm(myDownload(X,B2)),I) :-
        not isVIP(X), occurred(myDownload(X,B),I).
%rule 5 - revised
occurred(myDownload(X,B),I) :-
        occurred(download(X,Y,B),I),  holdsat(hasblock(Y,B),I).
%rule 7 - new
occurred(misuse(X),I) :-
        occurred(viol(myDownload(X,B)),I).
```

In summary, after a few iterations *rule 4* is corrected by adding an exception *not isVIP(X)*, *rule 5* is revised by correcting a typographical error in its condition (i.e. the name of a variable was not the intended one – *occurred(download(**Y**,Y,B),I)*), and finally, a new rule is learnt that defines *misuse* coherently with respect to the provided use case.

[2] The revision is generated in 23 seconds by ICLINGO (Gebser *et al*. 2007) on a 2.8 GHz Intel Core 2 Duo iMac with 4 GB of RAM.

https://doi.org/10.1017/S1471068411000305 Published online by Cambridge University Press

1 – Pre-processing (*rules in* $\overline{N_T}$)

```
terminated(perm(myDownload(X,B2)),I) :-
  try(4, 1, occurred(myDownload(X,B),I)),
  not exception(terminated(perm(
      myDownload(X,B2)),I), B).

try(4, 1, occurred(myDownload(X,B),I)) :-
  not del(4, 1),
  occurred(myDownload(X,B),I).

try(4, 1, occurred(myDownload(X,B),I)) :-
  del(4, 1).
```

2 – Learning (*rule in H*)

```
exception(terminated(perm(myDownload(X,B2
    )),I), B) :-
  isVIP(X).
```

3 – Postprocessing (*rule in* N'_T)

```
terminated(perm(myDownload(X,B2)),I) :-
  not isVIP(X),
  occurred(myDownload(X,B),I).
```

Fig. 3. Detailed revision transformations for *rule 4* (Section 4.2).

Input: N_B fixed theory; $N_T \in s(M)$ revisable theory; P set properties; M mode declarations
Output: N'_T revised theory according to the given P
$(\overline{N_T}, \overline{M}) =$ pre-processing(N_T, M);
$H =$ ASPAL$(P, N_B \cup \overline{N_T}, \overline{M})$;
$N'_T =$ post-processing(N_T, H);
return N'_T;

Algorithm 1: Phases of the revision algorithm.

5 Theory revision through ASP

In this section we provide more details about the revision process. We first introduce all the computational steps to derive a revision with respect to a set of use cases. Then we delve into the details of the learning system, describing the integrated ASP-based ILP approach.

The revised normative system $N_B \cup N'_T$ is computed by means of two program transformations and an abductive reasoning process executed in ASP, which derives prescriptions for revisions and new rules in the form of abducibles. The abductive solution has a one-to-one mapping to a revision of the initial theory.

5.1 Revision

The approach described in this section can be applied to other problems of TR. To the best of our knowledge, our methodology is the only one currently available that is able to support revision of non-monotonic *AnsProlog* theories that support integrity constraints, aggregates and other ASP constructs, providing revisions as answer sets. Operationally, the revision is performed using a similar transformation to the one described in Corapi *et al.* (2009). Figure 3 details the revision steps for one of the rules in the case study described above and Algorithm 1 illustrates the phases. We present the conceptual steps and refer the reader to Corapi *et al.* (2009) for further details.

A *pre-processing phase* lifts the standard ILP process of learning hypotheses about examples up to the (meta-)process of learning hypothesis about the rules and their exception cases. For every rule in N_T, every body literal c^i_j is replaced by the atom $try(i, j, c^i_j)$, where i is the index of the rule, j is the index of the body literal in the rule and the third argument is a reified term for the literal c^i_j. *not exception*(i, h_i, v_i) is added to the body of the rule where i is the index of the rule, h_i is the reified term for the head of the rule and v_i is an optional list of additional variables appearing in the body (see Figure 3). The *try* predicate

https://doi.org/10.1017/S1471068411000305 Published online by Cambridge University Press

is defined in such a way that whenever $del(i, j)$ is true, the meta-condition $try(i, j, c_j^i)$ is always true. Otherwise $try(i, j, c_j^i)$ is true whenever c_j^i is true. Facts of the type $del(i, j)$ can be learnt by the ILP system used within the revision. M specifies mode declaration of rules that can be added together with additional head declarations that are added to take into account the newly introduced *del* and *exception* predicates.

In the *learning phase*, given the pre-processed theory $\overline{N_T}$ and the new mode declarations $\overline{M}$, the following ILP task is executed $\langle P, N_B \cup \overline{N_T}, \overline{M} \rangle$ using ASPAL, the learning system described in Section 5.2. The outcome of the learning phase H is used in a *post-processing phase*, which generates a revised theory N_T' semantically equivalent to $\overline{N_T} \cup H$. Informally, for each $del(i, j)$ fact in H the corresponding condition j in rule i in N_T is deleted. For each exception rule in H of the form $exception(i, h_i, v_i) \leftarrow c_1, \ldots, c_n$, the corresponding rule i in N_T is substituted with n new rules, one for each condition c_h, $1 \leqslant k \leqslant n$. Each of these rules k will have in the head the predicate h_i and in the body all conditions present in the original rule i in N_T plus the additional condition *not* $c(k)$. An exception with empty body results in the original rule i being deleted. An exception for which at least two conditions share variables is kept as an additional 'exception concept' in the revised theory. The pre-processing and post-processing phases perform syntactic transformations that are answer set preserving and do not involve the answer set solver.

5.2 ASPAL

The system used in this work, called ASP Abductive Learning (ASPAL) , though used here to support the revision of a normative system, can be applied more generally to non-monotonic ILP problems. It is based on the transformation from an ILP task to an abductive reasoning task, used in a recently proposed ILP system (Corapi *et al*. 2010).

This system offers several advantages over other existing ILP approaches, making it particularly suited for normative design. ASPAL is able to handle negation within the learning process, and therefore reason about default assumptions governing inertial fluents; to perform non-observational and multiple predicate learning, thus computing hypotheses about causal dependencies between observed sequences of events and normative states and to learn non-monotonic hypothesis, which is also essential for theory revision. Furthermore, the learning can be enabled by a simple transformation of the mode declarations and does not require the computation of a *bridge theory* (Yamamoto *et al*. 2010). As discussed in Corapi *et al.* 2010, none of the existing ILP systems provides the above-mentioned features. Embedding the learning process within ASP reduces the semantic gap between the normative system and the learning process and permits an easier control of the whole process. The notion of revision distance as in Definition 3 can be managed by the optimisation facilities provided by modern ASP solvers (Gebser *et al*. 2007). Optimisation statements can be used to derive answer sets that contain a minimal number of atoms of a certain type that ultimately relate to new rules or revisions as explained in this section.

As in Corapi *et al.* (2010), an ILP task $\langle P, B, M \rangle$ is transformed into an abductive logic programming problem (Kakas *et al*. 1992), thus enabling the use of *AnsProlog*. Let us introduce some preliminary notation. Given a mode declaration $modeh(s)$ or $modeb(s)$, id is a unique identifier for mode declaration, $\mathbf{s}$ is the literal obtained from s by replacing all placemarkers with different variables $X_1, \ldots, X_n$; $type(\mathbf{s}, s)$ denotes the sequence of literals

https://doi.org/10.1017/S1471068411000305 Published online by Cambridge University Press

$t_1(X_1), \ldots, t_n(X_n)$ such that t_i is the type of placemarker replaced by variable X_i; $con(\mathbf{s}, s) = (C_1, \ldots, C_c)$ is the *constant list* of variables in $\mathbf{s}$ that replace only constant placemarkers in s. $inp(\mathbf{s}, s) = (I_1, \ldots, I_i)$ and $out(\mathbf{s}, s) = (O_1, \ldots, O_o)$ are defined similarly for input and output placemarkers. As s is clear from the context, in the following we omit the second argument from $type(\mathbf{s}, s)$, $con(\mathbf{s}, s)$, $inp(\mathbf{s}, s)$ and $out(\mathbf{s}, s)$.

Given a set of mode declarations M, a *top theory* $\top = t(M)$ is constructed as follows:

- For each head declaration $modeh(s)$, with unique identifier id, the following rule is in $\top$

$$\begin{aligned} &\mathbf{s} \leftarrow \\ &\quad rule(RId, (id, con(\mathbf{s}), ())), \\ &\quad rule\_id(RId), \\ &\quad type(\mathbf{s}), \\ &\quad body(RId, 1, inp(\mathbf{s})). \end{aligned} \tag{1}$$

- For each body declaration $modeb(s)$, with unique identifier id the following clause is in $\top$

$$\begin{aligned} &body(RId, L, I) \leftarrow \\ &\quad rule(RId, L, (id, con(\mathbf{s}), Links)), \\ &\quad link(inp(\mathbf{s}), I, Link), \\ &\quad \mathbf{s}, \\ &\quad type(\mathbf{s}), \\ &\quad append(I, out(\mathbf{s}), O), \\ &\quad body(RId, L + 1, O). \end{aligned} \tag{2}$$

- The following rule is in $\top$ together with the definitions for the *link*, *rule_id* and *append* predicates:

$$body(RId, L, \_) \leftarrow rule(RId, L, last) \tag{3}$$

$rule\_id(rid)$ is true whenever $1 \leqslant rid \leqslant rn$, where rn is the maximum number of new rules allowed. $link((a_1, \ldots, a_m), (b_1, \ldots, b_n), (o_1, \ldots, o_m))$ is true if for each element in the first list a_i, there exists an element in the second list b_j such that a_i unifies with b_j and $o_i = j$. Given the top theory, we seek a set of *rule* atoms Δ such that P is true for all models of $B \cup \top \cup \Delta$. Δ has a one-to-one mapping to a set of rules $H = u(\Delta, M)$. Intuitively, each abduced atom represents a literal of the rule labelled by the first argument. The second argument collects the constant used in the literal and the third disambiguates the variable linking. Figure 4 shows the learning steps for *rule 4* of our example.

For space limitations we only state the main soundness and completeness theorem (Corapi and Russo 2011) of the learning system.

Theorem 1

Given an ILP task $\langle P, B, M \rangle$, H is an inductive solution if and only if there is a Δ such that $H = u(\Delta, M)$, $\top = t(M)$ and P is true in all the answer sets of $B \cup \top \cup \Delta$.

The ASP solver is used to compute a set of solutions Δ that can be translated back into a set of inductive solutions. Soundness and completeness for the revision procedure rely on Theorem 1 and the underlying ASP solver properties. These properties also ensure

https://doi.org/10.1017/S1471068411000305 Published online by Cambridge University Press

Inputs

Mode declarations M

```
exception(terminated(perm(myDownload
  (+agent,+block)),+instant), +block).
```

Properties P

```
viol(myDownload(alice,x4),i06).
not viol(myDownload(A,B),T), T!= i06.
occurred(misuse(alice), i06).
not occurred(misuse(X), T), T!= i06.
```

Background theory B

```
terminated(perm(myDownload(X,B2)),I) :-
  try(4, 1, occurred(myDownload(X,B),I)),
  not exception(terminated(perm(
      myDownload(X,B2)),I), B).

try(4, 1, occurred(myDownload(X,B),I)) :-
  not del(4, 1),
  occurred(myDownload(X,B),I).

try(4, 1, occurred(myDownload(X,B),I)) :-
  del(4, 1).
```

Top theory $\top$

```
exception(4, terminated(perm(myDownload(A
    ,B)),T)) :-
  instant(T), block(B), agent(A),
  rule_id(RID),
  rule(RID, 0, (e4, (), ())),
  body(RID, 1, (A, B, T)).

body(RID, Level, (A, B, T)) :-
  agent(A), block(B), instant(T),
  rule_id(RID),
  link(L1, (A, B, T), LR1),
  rule(RID, Level, (isv, (), (LR1))),
  isVIP(L1),
  body(L + 1, RID, (A, B, T)).

body(RID, L, _):-
  rule(RID, L, last).
```

Abductive solution Δ

```
rule(0, 0, (e4, (), ())),
rule(0, 1, (isv, (), (1))),
rule(0, 2, last)
```

Output

Inductive solution *H*

```
exception(terminated(perm(myDownload
  (X,B2)),I), B) :-
  isVIP(X).
```

Fig. 4. Learning steps for *rule 4* (Section 4.2). We show only the relevant mode declarations and rules.

that if a set of theories that matches the requirements exists within the language bias of the learning, in the limit, if a complete set of all use cases (an extensional specification of the requirements) is provided, the revision converges to the expected theory. This is of course an ideal case. In practice the system outputs more accurate solutions as more comprehensive use case sets are provided.

6 Discussion and related work

The motivation behind this paper is the problem of how to converge upon a complete and correct normative system *with respect to the intended range of application*, where in practice these properties may be manifested by incorrect or unexpected behaviour in use. In addition, we observe from practical experience with our particular framework that it is often desirable 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 (García-Ojeda *et al*. 2007), Operetta (Okouya and Dignum 2008), InstAL (Cliffe *et al*. 2006), MOISE (Hübner *et al*. 2007), Islander (Esteva *et al*. 2002), OCeAN (Fornara *et al*. 2008) and the constraint approach of Garcia-Camino *et al.* (2009)) 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 Garion *et al.* (2009) that capture consistency and completeness

https://doi.org/10.1017/S1471068411000305 Published online by Cambridge University Press

via modalities and other formalisms like Boella *et al.* (2009b), which capture the concept of norm change, or Vasconcelos *et al.* (2007) and Cardoso and Oliveira (2008); (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 position of other agents (Artikis 2009; Christelis and Rovatsos 2009). Essentially, the objective of each of the above is to realize a transformation of the normative framework to accommodate some form of shortcomings. 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 (Kollingbaum *et al.* 2007) 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 (Savarimuthu and Cranefield 2009) 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 the 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 (in press), where they propose the specification of incomplete theories and their management through incomplete normative states identified as 'pending'.

In Boella *et al.* (2009c), 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 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 Boella *et al.* (2009c) 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. Within the context of software engineering (Alrajeh *et al*. 2007) shows 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 an envisioned system and its environment, to compute missing requirement specifications. There are several elements in common with the scheme proposed here.

From an ILP perspective, we employ a system that can learn logic programs with negation (stratified or otherwise) and, unlike other existing nonmonotonic ILP systems (Sakama 2001b), is supported by completeness results, is integrated into ASP and can be tailored to particular design requirements. Some properties and results of ILP in the context of ASP are shown by Sakama (2010a). The author also proposes an algorithm for learning that is sound but not complete and, differently from the approach proposed here, employs a covering loop approach.

7 Conclusions and future work

The motivation for this work stems from a real need for tool support in the design of normative frameworks, because, although high-level, it is nevertheless hard for humans

https://doi.org/10.1017/S1471068411000305 Published online by Cambridge University Press

to identify errors in specifications, or indeed to propose the most appropriate corrective actions. We have described a methodology for the revision of normative frameworks and how to use tools with formal underpinnings to support the process. Specifically, we are able to revise a formal model – represented as a logic program – that captures the rules of a normative system. The revision is achieved by means of an ILP, working with the same representation, informed by use cases that describe instances of expected behaviour of the normative system. If actual behaviour does not coincide with the expected one, theory revision proposes new rules, or modifications of existing rules, for the normative framework. Furthermore, given correct traces, the learning process guarantees convergence – the property of 'learning in the limit'.

From this firm foundation, which properly connects a theory of normative systems with a practical representation, there are three directions that we aim to pursue: (i) definition of criteria for selecting solutions from alternative suggestions provided by the learning (we are currently investigating the use of *crucial literals* (Sattar and Goebel 1991)); (ii) introduction of levels of confidence in the use cases and their use for selecting the 'most likely' revision, in addition to the general criteria of minimal revision, i.e. combine some domain-independent heuristics with some domain-specific heuristics such as level of confidence in use cases and (iii) extension to interactions between normative frameworks and a form of cooperative revision. In addition, there is the matter of scalability. The computation time increases with the number of rules, time steps, errors in the theory and, in particular, mode declarations and language bias for the learning. That is, it grows with the state space of the normative framework and the 'learning space', i.e. is all possible theories we can construct given our language bias. We need to experiment further to understand better to which factors performance is sensitive and how to address these issues.

References

ALRAJEH, D., RAY, O., RUSSO, A. AND UCHITEL, S. 2007. Extracting requirements from Scenarios using ILP. In *Lecture Notes in Artificial Intelligence*, S. Muggleton, R. P. Otera, and A. Tamaddoni-Nezhad, Eds. Vol. 4455, Springer-Verlag, New York, USA, 63–77.

ARTIKIS, A. 2009. Dynamic protocols for open agent systems. In *Proceedings of International Conference on Agents and Multi-Agent Systems* (AAMAS), Decker, Sichman, Sierra and Castelfranchi, Eds., May, 10–15, 2009, Budapest, Hungary, 97–104.

BOELLA, G., NORIEGA, P., PIGOZZI, G. AND VERHAGEN, H., Eds. 2009a. Normative mult-agent systems, number 09121. In *Dagstuhl Seminar Proceedings*, Schloss Dagstuhl, Germany.

BOELLA, G., PIGOZZI, G. AND VAN DER TORRE, L. 2009b. Normative framework for normative system change. See Sierra *et al*. (2009), 169–176.

BOELLA, G., PIGOZZI, G. AND VAN DER TORRE, L. 2009c. Normative systems in computer science – ten guidelines for normative multiagent systems. See Boella *et al.* (2009a).

CARDOSO, H. L. AND OLIVEIRA, E. C. 2008. Norm defeasibility in an institutional normative framework. In *European Conference on Artificial Intelligence* (ECAI), M. Ghallab, C. D. Spyropoulos, N. Fakotakis, and N. M. Avouris, Eds. Frontiers in Artificial Intelligence and Applications, Vol. 178, IOS, Virginia, USA, 468–472.

CHRISTELIS, G. AND ROVATSOS, M. 2009. Automated norm synthesis in an agent-based planning environment. See Sierra *et al*. (2009), 161–168.

https://doi.org/10.1017/S1471068411000305 Published online by Cambridge University Press

CLIFFE, O. 2007. *Specifying and Analysing Institutions in Multi-Agent Systems Using Answer Set Programming*. Ph.D. thesis, University of Bath, North East Somerset, UK.

CLIFFE, O., DE VOS, M. AND PADGET, J. 2006. Answer set programming for representing and reasoning about virtual institutions. In *Seventh International Workshop on Computational Logic in Multi-Agent Systems* (CLIMA VII). Lecture Notes in Artificial Intelligence (LNAI), Vol. 4371. Springer, New York, USA, 60–79.

CORAPI, D., RAY, O., RUSSO, A., BANDARA, A. K. AND LUPU, E. C. 2009. Learning rules from user behaviour. In *Artificial Intelligence Applications & Innovations* (AIAI), Vol. 296, Springer, Boston, 459–468.

CORAPI, D. AND RUSSO, A. 2011. *Aspal. Proof of Soundness and Completeness*. Technical Report DTR11-5, Department of Computing, Imperial College, London.

CORAPI, D., RUSSO, A. AND LUPU, E. 2010. Inductive logic programming as abductive search. In *Technical Communications of the 26th International Conference on Logic Programming*, M. Hermenegildo and T. Schaub, Eds. LIPICS, Vol. 7, Dagstuhl, Germany, 54–63.

ESTEVA, M., DE LA CRUZ, D. AND SIERRA, C. 2002. Islander: An electronic institutions editor. In AAMAS. ACM, 1045–1052.

FORNARA, N., VIGANÒ, F., VERDICCHIO, M. AND COLOMBETTI, M. 2008. Artificial institutions: A model of institutional reality for open multiagent systems. *Artificial Intelligence Law 16*(1), 89–105.

GARCÍA-CAMINO, A., RODRÍGUEZ-AGUILAR, J. A., SIERRA, C. AND VASCONCELOS, W. W. 2009. Constraint rule-based programming of norms for electronic institutions. *Autonomous Agents and Multi-Agent Systems 18*(1), 186–217.

GARCÍA-OJEDA, J. C., DELOACH, S. A., ROBBY, OYENAN, W. H. AND VALENZUELA, J. 2007. O-mase: A customizable approach to developing multiagent development processes. In *AOSE*, M. Luck and L. Padgham, Eds. Lecture Notes in Computer Science, Vol. 4951. Springer, New York, USA, 1–15.

GARION, C., ROUSSEL, S. AND CHOLVY, L. 2009. A modal logic for reasoning on consistency and completeness of regulations. See Boella *et al*. (2009).

GEBSER, M., KAUFMANN, B., NEUMANN, A. AND SCHAUB, T. 2007. clasp: A conflict-driven answer set solver. In *LPNMR'07*. Springer, New York, USA, 260–265.

GELFOND, M. AND LIFSCHITZ, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing 9*(3–4), 365–386.

GELFOND, M. AND LIFSCHITZ, V. 1998. Action languages. *Electronic Transactions Artificial Intelligence 2*, 193–210.

HÜBNER, J. F., SICHMAN, J. S. AND BOISSIER, O. 2007. Developing organised multiagent systems using the moise. *International Journal of Agent-Oriented Software Engineering 1*(3/4), 370–395.

JONES, A. J. AND SERGOT, M. 1996. A formal characterisation of institutionalised power. *ACM Computing Surveys 28*(4es), 121. (Read 28/11/2004).

KAKAS, A. C., KOWALSKI, R. A. AND TONI, F. 1992. Abductive logic programming. *Journal of Logic Computer 2*(6), 719–770.

KOLLINGBAUM, M., NORMAN, T., PREECE, A. AND SLEEMAN, D. 2006. Norm conflicts and inconsistencies in virtual organisations. In *Proceedings of COIN 2006*, 245–258.

KOWALSKI, R. AND SERGOT, M. 1986. A logic-based calculus of events. *New General Computer 4*(1), 67–95.

MUGGLETON, S. 1995. Inverse entailment and progol. *New General Computer 13*(3&4), 245–286.

OKOUYA, D. AND DIGNUM, V. 2008. Operetta: A prototype tool for the design, analysis and development of multi-agent organizations. In *AAMAS (Demos)*. IFAAMAS, 1677–1678.

SAKAMA, C. 2001a. Learning by answer sets. In *Proceedings of the AAAI Spring Symposium on Answer Set Programming*, 181–187, AAAI Press, California, USA.

SAKAMA, C. 2001b. Nonmonotonic inductive logic programming. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning* (LPNMR 6). Notes in Artificial Intelligence 2173, Springer-Verlag, Berlin, Germany, 62–80.

SATTAR, A. AND GOEBEL, R. 1991. Using crucial literals to select better theories. *Computational Intelligence 7*, 11–22.

SAVARIMUTHU, B. T. R. AND CRANEFIELD, S. 2009. A categorization of simulation works on norms. See Boella *et al.* (2009a).

SERRANO, J.-M. AND SAUGAR, S. 2009. Dealing with incomplete normative states. In *Proceedings of COIN 2009*, J. A. Padget, A. Artikis, W. W. Vasconcelos, K. Stathis, V. Torres da Silva, E. T. Matson, and A. Polleres, Eds. LNCS, Vol. 6069. Springer, New York, USA, 304–319.

SIERRA, C., CASTELFRANCHI, C., DECKER, K. S. AND SICHMAN, J. S., Eds. 2009. *AAMAS 2009*, Budapest, Hungary, May 10–15, 2009, Vol. 1. IFAAMAS.

VASCONCELOS, W., KOLLINGBAUM, M. AND NORMAN, T. 2007. Resolving conflict and inconsistency in norm-regulated virtual organizations. In *AAMAS*, E. H. Durfee, M. Yokoo, M. N. Huhns, and O. Shehory, Eds. IFAAMAS, 91.

WOGULIS, J. AND PAZZANI, M. J. 1993. A methodology for evaluating theory revision systems: Results with audrey ii. In *International Joint Conference on Artificial Intelligence* (IJCAI), 1128–1134.

YAMAMOTO, Y., INOUE, K. AND IWANUMA, K. 2010. From inverse entailment to inverse subsumption. In *20th International Conference on Inductive Logic Programming* (ILP), Firenze, Italy, June 27–30.