Handling Change in Normative Specifications

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

[1] Department of Computing
Imperial College London
{da407,d.corapi,a.russo}@imperial.ac.uk
[2] Department of Computing
University of Bath
{mdv,jap}@cs.bath.ac.uk
[3] Principles of Informatics Research Division
National Institute of Informatics
ksatoh@nii.ac.jp

Abstract. Normative frameworks provide a means to address the governance of open systems, offering a mechanism to express responsibilities and permissions of the individual participants with respect to the entire system without compromising their autonomy. In order to meet requirements careful design is crucial. Tools that support the design process can be of great benefit. In this paper, we describe and illustrate a methodology for elaborating normative specifications. We utilise use-cases to capture desirable and undesirable system behaviours, employ inductive logic programming to construct elaborations, in terms of revisions and extensions, of an existing (partial) normative specification and provide justifications as to why certain changes are better than others. The latter can be seen as a form of impact analysis of the possible elaborations, in terms of critical consequences that would be preserved or rejected by the changes. The main contributions of this paper is a (semi) automated process for controlling the elaboration of normative specifications and a demonstration of its effectiveness through a proof-of-concept case study.

1 Introduction

Normative frameworks provide a powerful tool for governing open systems by providing guidelines for the behaviour of the individual components without regimentation [1]. Using a formal declarative language to specify the behaviour of a normative system gives the system's designer a means to verify the compliance of the system with respect to desirable behaviours or properties [2, 3]. When errors are detected, manually identifying what changes to make in order to attain compliance with desired behaviours is often difficult and error-prone: additional errors may be inadvertently introduced in the specification as a result of misinterpretations, incompleteness or unexpected impact of the manual changes. The availability of a systematic and automated framework for elaborating and handling change in normative specifications would benefit the development process of such systems.

Corapi et al. [4] have shown how Inductive Logic Programming (ILP) can be used to support the elaboration of partial normative specifications, modelled using Answer Set Programming (ASP). The system designer provides intended behaviours in the form of use-cases. These are defined as specific (partial) scenarios of events and expected outcomes, and are used to validate the correctness of the specifications. Use-cases that fail the validation process are taken as positive examples (or *learning objectives*) for an inductive learning tool, which in turn constructs suggestions for improving the specification to guarantee the satisfiability of the failed use-cases. The learning of such suggestions (or *elaborations*) is performed within a search space defined by a given set of mode declarations that captures the format of possible changes that could be applied to a given formalised normative specification.

Use-cases are inherently partial descriptions of a system behaviour. While their sparse nature is well suited for the non-monotonicity of ASP, the learning process also becomes less restricted, thus causing the problem of *how to choose* among the multiple suggestions for change computed by the learner. For example, the failure to signal a violation when an agent tries to borrow a book from a library could be caused by the specification not correctly capturing any one of the following conditions: (i) the agent has already borrowed the maximum number of items allowed, (ii) the book is for reference only, or (iii) a combination of all these reasons. In general, to address any of these errors and establish the desired violations, there is more than one possible revision for the given specification, with each one having its own impact on the overall behaviour of the system. Thus, the problem with choosing the most appropriate revision is not the revision itself, but the effect of that revision when it is combined with the rest of the system and ensuring that desired system properties are maintained and undesired ones are not introduced.

The approach in [4] lacks criteria for selecting among a (possibly large) number of learned suggestions. This paper addresses this limitation and the general problem of how to choose between alternative changes to make to a (partial) normative specification, by providing an approach for analysing the impact of these changes. We make use of the notion of *relevant literals* as critical elements of the domain that are required to be positive or negative consequences in the intended specification, in order to discriminate between the suggested changes. The solution proposed in this paper provides also a general method for choosing among alternative hypotheses in the wider context of inductive learning.

The remainder of the paper is structured as follows: the next two sections provide background in the form of a summary of the formal and computational model (section 2) and an outline of the revision process (section 3) as described in detail in [4]; the method of test generation and the ranking of results is described in section 4 and then demonstrated in section 5 using the same file-sharing scenario as [4]. The paper ends with a discussion of some related work (section 6) and conclusions (section 7).

2 Normative Framework

Actions that we take in society are regulated by laws and conventions. Similarly, actions taken by agents or entities in open systems may be regulated or governed by the social

rules of the system they operate in. It is the task of the normative framework to specify these rules and observe the interactions between the various entities with respect to these rules. The essential idea of normative frameworks is a (consistent) collection of rules whose purpose is to describe *A standard or pattern of social behaviour that is accepted in or expected of a group [OED]*. These rules may be stated in terms of events or actions, but specifically the events that matter for the functioning of the normative framework, based on its current state. In turn, each event/action can influence the normative state.

The control of an agent's or entity's power (effectiveness of an action) and permission to perform certain actions, its obligations and violations of the norms, needs to occur within the context of normative system. For example, raising a hand in class means something different than raising a hand during an auction. This relation between the physical and normative context is described by *Conventional Generation* [5] where an event in the physical world may correspond to an *normative event*. An example is clicking the "buy" button on Amazon, which *counts as* paying for the good.

2.1 The Formal Model

In this paper we use the model as set out in [2] based on the concept of *exogenous events* within the physical world and *normative states*, those within the framework's context. Events change the state of the normative system by acting on *normative fluents*, properties of the system that can be true at certain points in time.

The essential elements of the normative framework (summarised in Fig. 1(a)) are events ($\mathcal{E}$), which bring about changes in state, and fluents ($\mathcal{F}$), 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. The model has two kinds of events: normative ($\mathcal{E}_{norm}$), that are the events defined by the framework, and exogenous ($\mathcal{E}_{ex}$), some of whose occurrence may trigger normative events in a direct reflection of "counts-as" [6], while the rest may have no relevance for a given framework. Normative events are further partitioned 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, (i.e. from the occurrence of a non-permitted event), or from the non-fulfilment of an obligation. The model also has 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 a particular normative framework.

A normative state is represented by the fluents that hold true in that state. Fluents that are not present are held to be false. Conditions on a state ($\mathcal{X}$) are expressed by a set of fluents that should be true or false. The normative framework is initialised with the state $\mathcal{I}$.

Changes in the normative state are specified by two relations: (i) the generation relation ($\mathcal{G}$), 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 ($\mathcal{C}$), which specifies the initiation and termination of fluents, given a certain state condition and event.

$\mathcal{N} = \langle \mathcal{E}, \mathcal{F}, \mathcal{C}, \mathcal{G}, \mathcal{I} \rangle$, 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
   $C(X, e) =$
   $(\mathcal{C}^{\uparrow}(\phi, e), \mathcal{C}^{\downarrow}(\phi, e))$ where
   (i) $\mathcal{C}^{\uparrow}(\phi, e)$ initiates fluents
   (ii) $\mathcal{C}^{\downarrow}(\phi, e)$ terminates fluents
4. $\mathcal{E} = \mathcal{E}_{ex} \cup \mathcal{E}_{norm}$
   with $\mathcal{E}_{norm} = \mathcal{E}_{act} \cup \mathcal{E}_{viol}$
5. $\mathcal{I}$, initial instiutional state
6. State Formula: $\mathcal{X} = 2^{\mathcal{F} \cup \neg \mathcal{F}}$

(a)

$$p \in \mathcal{F} \Leftrightarrow \texttt{ifluent(p)}. \quad (1)$$
$$e \in \mathcal{E} \Leftrightarrow \texttt{event(e)}. \quad (2)$$
$$e \in \mathcal{E}_{ex} \Leftrightarrow \texttt{evtype(e, obs)}. \quad (3)$$
$$e \in \mathcal{E}_{act} \Leftrightarrow \texttt{evtype(e, act)}. \quad (4)$$
$$e \in \mathcal{E}_{viol} \Leftrightarrow \texttt{evtype(e, viol)}. \quad (5)$$
$$\mathcal{C}^{\uparrow}(\phi, e) = P \Leftrightarrow \forall \texttt{p} \in P\ \texttt{initiated(p, T)} :- \quad (6)$$
$$\texttt{occurred(e, I)}, EX(\phi, T). \quad (7)$$
$$\mathcal{C}^{\downarrow}(\phi, e) = P \Leftrightarrow \forall \texttt{p} \in P\ \texttt{terminated(p, T)} :- \quad (8)$$
$$\texttt{occurred(e, T)}, EX(\phi, T). \quad (9)$$
$$\mathcal{G}(\phi, e) = E \Leftrightarrow \forall \texttt{g} \in E, \texttt{occurred(g, T)} :- \texttt{occurred(e, T)}, \texttt{holdsat(pow(e), T)}, EX(\phi, T). \quad (10)$$
$$p \in \mathcal{I} \Leftrightarrow \texttt{holdsat(p, i00)}. \quad (11)$$

(b)

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

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 the eventual 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 non-permitted events and non-fulfilled obligations 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 determining the next state. For each trace, the normative framework can determine a sequence of states that constitutes the model of the framework for that trace. This process is realised as a computational process using answer set programming.

2.2 Computational Model

The formal model described above is translated into an equivalent computational model using answer set programming (ASP) [7] with $AnsProlog$ as the implementation language[4]. $AnsProlog$ is a knowledge representation language that allows the programmer to describe a problem and the requirements for solutions declaratively, rather than specifying an algorithm to find the solutions to the problem. The mapping follows the naming convention used in the Event Calculus [8] and Action languages [9].

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* or classical negation. *Literals* are atoms `a` or classically negated atoms $-$`a`. Extended literals are literals `l` or negated literals not `l`. The latter is true if there is no evidence supporting

[4] In this paper we use the SMODELS syntax for writing $AnsProlog$ programs

the truth of a. Atoms and (extended) literals are used to create rules of the general form: $\mathtt{a} :- \mathtt{b}_1, ..., \mathtt{b}_m, \text{not } \mathtt{c}_1, ..., \text{not } \mathtt{c}_n$, where a, $\mathtt{b}_i$ and $\mathtt{c}_j$ are literals. Intuitively, this means *if all literals* $\mathtt{b}_i$ *are known/true and no literal* $\mathtt{c}_j$ *is known/true, then* a *must be known/true*. a is called the head and $\mathtt{b}_1, ..., \mathtt{b}_m, \text{not } \mathtt{c}_1, ..., \text{not } \mathtt{c}_n$ the body of the rule. Rules with an empty body are called *facts*. Rules with an empty head are called *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*, that is, 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. They are computed by a program called an *answer set solver*. For this paper the solver we used was ICLINGO [10].

The mapping of a normative framework consists of two parts: an independent *base 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 (un)fulfilled obligations.

The mapping uses the following atoms:

- $\mathtt{ifluent}(\mathtt{p})$ to identify fluents,
- $\mathtt{evtype}(\mathtt{e}, \mathtt{t})$ to describe the type of an event,
- $\mathtt{event}(\mathtt{e})$ to denote the events,
- $\mathtt{instant}(\mathtt{i})$ for time instances,
- $\mathtt{final}(\mathtt{i})$ for the last time instance,
- $\mathtt{next}(\mathtt{i1}, \mathtt{i2})$ to establish time ordering,
- $\mathtt{occurred}(\mathtt{e}, \mathtt{i})$ to indicate that the (normative) event happened at time i,
- $\mathtt{observed}(\mathtt{e}, \mathtt{i})$ that the (exogenous) event was observed at time i,
- $\mathtt{holdsat}(\mathtt{p}, \mathtt{i})$ to state that the normative fluent p holds at i, and finally
- $\mathtt{initiated}(\mathtt{p}, \mathtt{i})$ and $\mathtt{terminated}(\mathtt{p}, \mathtt{i})$ for fluents that are initiated and terminated at i.

Given that exogenous events are always empowered while normative events are not, the mapping must keep type information for the events, hence the $\mathtt{evtype}(\mathtt{e}, \mathtt{t})$ atoms. Similarly, violation events are always permitted and empowered. However, all fluents, irrespective of type, are treated the same way so the mapping does not differentiate between them.

Figure 1(b) provides the framework-specific translation mechanism.An expression ϕ in the framework is translated into $AnsProlog$ rule bodies as conjunction of literals, using negation as failure for negated expressions, denoted as $EX(\phi, T)$. The translation of the formal model is augmented with a trace program that specifies (i) the length of traces that the designer is interested in, and (ii) the property that each, except the final, time instant is associated with exactly one exogenous event (iii) specifics of the desired trace(s), for example length, or the occurrence of a specific event.

3 Revising Normative Rules

In this section we briefly summarise the approach described in [4] for computing elaborations of normative specifications through use-cases by means of non-monotonic in-

ductive logic programming. Our proposed technique for analysing the impact that possible elaborations could have on a normative specification extends this approach with a formal mechanism for narrowing down the number of suggested elaborations based on a notion of *relevant literals*.

The development of a normative specification is captured in [4] by an iterative process that supports automated synthesis of new rules and revisions of existing one from given use-cases. The latter represent instances of executions that implicitly capture the desired behaviour of the system. They are defined as tuples $\langle T, O\rangle$ where T (trace) specifies a (partial) sequence of exogenous events ($\texttt{observed}(\texttt{e},\texttt{t})$), and O (output) describes the expected output as a set of `holdsat` and `occurred` literals that should appear in the normative state. The traces do not have to be complete (i.e. include an event for each time instance) and the expected output may contain positive as well as negative literals and does not have to be exhaustive. An existing (partial) normative specification N is validated against a use-case $\langle T, O\rangle$, specified by the designer, by using T as a trace program for N and adding a constraint that no answer set should be accepted that does not satisfy O. If no answer set is computed then the normative specification does not comply with the use-case and a learning step is performed to compute new rules and/or revisions of existing rules that guarantee the satisfiability of the use-case. This validity check can be extended to a set of use-cases U from which we derive the conjunction of all the traces T_U and outputs O_U (making sure that there is no conflict in the time points being used).

The learning step is in essence a Theory Revision [11] task, defined in terms of a non-monotonic inductive logic programming [12], and implemented in answer set programming using the learning system ASPAL [13], [14].

Within the context of our computational model of normative systems, this task is expressed as a tuple $\langle O_{\mathrm{U}}, N_{\mathrm{B}} \cup T_{\mathrm{U}}, N_{\mathrm{T}}, M\rangle$, where:

1. O_{U} is the set of expected outputs,
2. N_{B} is the part of the normative specification that is not subject to revisions (i.e. "static" background knowledge) augmented with the traces of the use-cases,
3. N_{T} is the part of the normative system that is subject to modification, and
4. M is the set of mode declarations that establish how rules in the final solution shall be structured. A mode declaration can be of the form $modeh(s)$ or $modeb(s)$, where s is the schema of the predicate that can be used in the head or body of a rule respectively.

These last define the literals that can appear in the head and in the body of a well-formed revision. The choice of the M is therefore crucial. Larger M with more mode declarations ensures higher coverage of the specification but increase the computation time. Conversely, smaller mode declarations improve performance but may result in partial or incorrectly formed solutions.

In [4] the mode declaration M is specified to allow the synthesis of new normative rules as well as revision of existing rules. To compute the first type of solutions, M allows predicates `occurred`, `initiated` and `terminated` to appear in the head of the learned rules and predicates `holdsat` and `occurred` to appear in the body of the learned rules. To compute revisions on existing rules the mode declaration M makes use of special predicates: $\texttt{exception}(\texttt{p},\bar{\texttt{v}})$, where `p` is a reified term for a rule existing

in the specification and $\bar{\mathtt{v}}$ the list of variables in the rule that are involved in the change. This special predicate can appear in the head of a learned rule whose body gives the new literals that need to be added to the existing rule `p` with specific variables $\bar{\mathtt{v}}$. Another special predicate is $\mathtt{del}(\mathtt{i},\mathtt{j})$, where `i` is the index of an existing rule and `j` the number of the literal in the body of the existing rule that needs to be removed. This is learned as a ground fact. By means of these two special predicates it is possible to learn rules that define what literals to add to and what literals to remove from existing rules of the normative specification N_{T}. The reader may refer to [4] for further details.

4 Handling Change

The approach proposed by [4] provides an automated way for computing suggestions of possible elaborations of a given normative specification. The designer must then choose the most appropriate revision from a (possibly large) set of alternative changes. In real applications this is impractical, as the number of suggested changes can be too large to work with. Informally, possible alternative revisions can be any combinations of addition of new literals and/or deletion of existing literals in any of the existing rules of the specification. Automated criteria for selecting solutions from the suggestions provided by the learning are therefore essential.

In the remainder of this paper, we show that analysing the impact of suggested changes, in terms of relevant literals that would be preserved or discarded, can be an effective criteria for revision selection. Considering all the consequences that each possible revision would give is clearly not a practical solution. What is needed is a mechanism for identifying key consequences that would allow to reject some suggested changes whilst preserving others. We propose that test generation can provide such a mechanism and show how the process can carried out in answer set programming to fit with both the inductive learner and the computational model of the normative frameworks.

4.1 Test Generation

A test normally defines the set of outcomes that have to be observed given certain *achievable information* in order to confirm or refute an hypothesis. Using the definitions from [15], a test can formally be defined as a pair (A, l) where A is a conjunction of achievable literals, the initial condition specified by the tester, and l is an observable, the outcome (l or $\neg l$) decided by the tester. Using this structure, we can define confirmation and refutation tests with respect to given background knowledge Σ.

Definition 1. *The outcome a of a test is said to* confirm *a hypothesis H iff $\Sigma \wedge A \wedge H$ is satisfiable and $\Sigma \wedge A \vDash H \rightarrow a$. The outcome a of a test is said to* refute *a hypothesis H iff $\Sigma \wedge A \wedge H$ is satisfiable and $\Sigma \wedge A \vDash H \rightarrow \neg a$.*

Hence, a refutation test has the power to eliminate the hypothesis when its outcome is not included in the consequence of $\Sigma \wedge A$ where H is true. Note that in this paper the

symbol $\models$ is associated with the *skeptical stable model* semantics[5] in conformity with the underlying ASP framework.

Using the notion of relevant test in [15], we define *relevant literals* as follows.

Definition 2. *(*Relevant Literal*) Let $\langle T, O\rangle$ be a use-case consisting of a partial trace T and desired outcome O, Σ a given (partial) normative specification, and HYP the set of hypotheses representing the suggested revisions of Σ that satisfy $\langle T, O\rangle$. A literal l is* relevant *if:*

1. $\Sigma \wedge T \wedge O \wedge H_i$ *is satisfiable, for all* $H_i \in HYP$
2. $\Sigma \wedge T \wedge O \not\models \bigvee_{H_i \in HYP} \neg H_i$
3. $T \wedge O \wedge l$ *is an abductive explanation for* $\bigvee_{H_i \in HYP} \neg H_i$
4. $T \wedge O \wedge l$ *is not an abductive explanation for* $\neg H_i$*, for all* $H_i \in HYP$

Conditions 1 and 2 above state, respectively, that each suggested revision (H_i) satisfies the given use-case and is consistent with the normative specification and the use-case. Both these conditions are guaranteed by the correctness of the learning process [13]. Conditions 3 and 4 above ensure that some but not all suggested revisions are refuted by the relevant literal l. Thus should l be observed, at least one hypothesis may be rejected.

The automated generation of tests for specific objectives (e.g. eliminate some hypothesis H) can be formulated [15] in terms of an abductive problem [16] so that $\Sigma \cup (A, l) \models \neg H$. Informally, given an abductive problem $\langle B, Ab, G\rangle$, where B is a background knowledge, G is a goal, and Ab a set of abducibles, a conjunction of literals E in the language Ab, is an *abductive explanation* for G, with respect to B if and only if $B \wedge E$ is satisfiable and $B \wedge E \models G$.

4.2 The approach

Our approach extends the work of [4] with an iterative process for computing relevant literals and discarding learned revisions that are refuted by the relevant literals. As illustrated in Fig. 2, once possible changes are learned, this iterative process is activated. At each iteration, the (remaining) learned revisions are "combined" with the existing normative specification as integrity constraints in order to capture conditions 3 and 4 above and ensure that the abduced relevant literals have the power to eliminate some suggested revisions. Traces of the given use-cases are included as achievable literals to guarantee that the abduced relevant literals conform with the use-cases. The abduced relevant literals are ranked according to how much information can be gained from them. The most highly ranked literal is then presented to the designer, who can then specify the truth value for the literal. Based on the designer's answer, suggested revisions that are refuted by the relevant literal are discarded. The process is repeated: new relevant literals and their scores are computed with respect to the remaining suggested revisions. This process is repeated until no further relevant literals can be identified. This is the inner loop of the process depicted in Fig. 2. The remaining learned revisions are then returned to the designer. If only one suggested revision remains, this is used to change the specification automatically and the revised normative description is returned.

[5] $P \models a$ if a is true in every answer set of P.

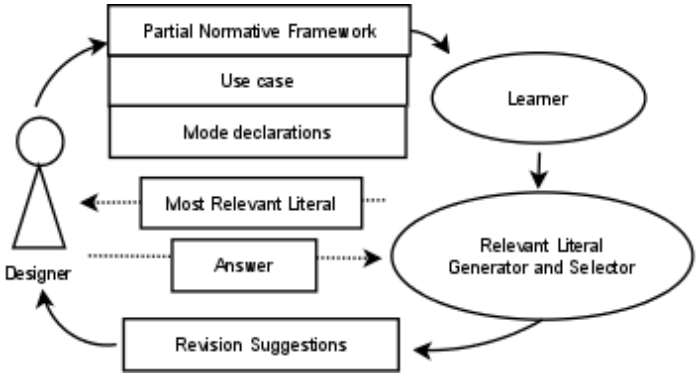

Fig. 2. Handling changes in normative specifications

Suggested Revisions as Hypotheses. Changes to our normative specifications can be one of three different varieties: addition of new rules, deletion of an existing rule, and addition or deletion of a body literal in an existing rule. These modifications correspond to the following facts in each solution:

1. $r \leftarrow c_1, \ldots, c_n$: A new rule is added to the revised specification.
2. $del(i, j)$: The condition j of rule r_i in N_{T} is deleted. If a rule has all of its condition deleted, then it is removed from the revised specification.
3. $xt(i, r_i) \leftarrow c_1, \ldots, c_n$: The condition of rule r_i in N_{T} is extended with the conditions $c_1, \ldots, c_n$. Should a solution contain two of such facts for extending the same rule, then the revised specification contains two different versions of the extended rule.

To abduce relevant literals, each modification in a learned solution is (automatically) combined with the static part of the background knowledge N_{B}. For each revisable rule r_i in solution S_k the following clause is added:

1. If r_i is deleted by S_k, then clauses corresponding to r_i are not added to N_{B}
2. $\neg \mathtt{hyp_k} :- \mathbf{not}\ \mathtt{r_i}, c_1, \ldots, c_n, c_{n+1}, \ldots, c_m$
   If both $xt(i, r_i) \leftarrow c_1, \ldots, c_n$ and $del(i, j)$ facts are in S_k and $c_{n+1}, \ldots, c_m$ are the conditions of rule r_i from N_{T} that are not deleted by S_k
3. $\neg \mathtt{hyp_k} :- \mathbf{not}\ \mathtt{r_i}, c_1, \ldots, c_m$
   If only $del(i, j)$ is in S_k, and $c_1, \ldots, c_m$ are conditions of rule r_i from N_{T} that are not deleted by S_k
4. $\neg \mathtt{hyp_k} :- \mathbf{not}\ \mathtt{r_i}, c_1, \ldots, c_n, c_{n+1}, \ldots, c_m$
   If only $xt(i, r_i) \leftarrow c_1, \ldots, c_n$ is in S_k, and $c_{n+1}, \ldots, c_m$ are the conditions of r_i from N_{T}
5. $\neg \mathtt{hyp_k} :- \mathbf{not}\ \mathtt{r_i}, c_1, \ldots, c_m$
   If S_k does not change r_i, and $c_1, \ldots, c_m$ are the conditions of r_i from N_{T}

For example, if we have the following N_{T}:

$\texttt{terminated}(\texttt{perm}(\texttt{shoot}(\texttt{A1}, \texttt{A2})), \texttt{Time}) : -\texttt{initiated}(\texttt{peace}, \texttt{Time}).$
$\texttt{terminated}(\texttt{perm}(\texttt{shoot}(\texttt{A1}, \texttt{A2})), \texttt{Time}) : -\texttt{holdsat}(\texttt{peace}, \texttt{Time}).$

...and three alternative suggested revisions:

1. Add: $\texttt{initiated}(\texttt{perm}(\texttt{shoot}(\texttt{A1}, \texttt{A2})), \texttt{Time}) : -\texttt{initiated}(\texttt{war}, \texttt{Time})$. The following rules are added to the normative specification, with head predicate $\neg\texttt{hyp}(1)$:

$$\begin{aligned}
\neg\texttt{hyp}(1) : - \ & \textbf{not } \texttt{initiated}(\texttt{perm}(\texttt{shoot}(\texttt{A1}, \texttt{A2})), \texttt{Time}), \\
& \texttt{initiated}(\texttt{war}, \texttt{Time}). \\
\neg\texttt{hyp}(1) : - \ & \textbf{not } \texttt{terminated}(\texttt{perm}(\texttt{shoot}(\texttt{A1}, \texttt{A2})), \texttt{Time}), \\
& \texttt{initiated}(\texttt{peace}, \texttt{Time}). \\
\neg\texttt{hyp}(1) : - \ & \textbf{not } \texttt{terminated}(\texttt{perm}(\texttt{shoot}(\texttt{A1}, \texttt{A2})), \texttt{Time}), \\
& \texttt{holdsat}(\texttt{peace}, \texttt{Time}).
\end{aligned}$$

   The first of the above rules represents the new rule added by the suggestion, while the latter two correspond to changes made by alternative revision suggestions but left unchanged by the current suggestion.

2. Change: $\texttt{terminated}(\texttt{perm}(\texttt{shoot}(\texttt{A1}, \texttt{A2})), \texttt{Time}) : -\texttt{initiated}(\texttt{peace}, \texttt{Time})$. to: $\texttt{terminated}(\texttt{perm}(\texttt{shoot}(\texttt{A1}, \texttt{A2})), \texttt{Time}) : -\texttt{terminated}(\texttt{war}, \texttt{Time})$. The following rules are added:

$$\begin{aligned}
\neg\texttt{hyp}(2) : - \ & \textbf{not } \texttt{terminated}(\texttt{perm}(\texttt{shoot}(\texttt{A1}, \texttt{A2})), \texttt{Time}), \\
& \texttt{terminated}(\texttt{war}, \texttt{Time}). \\
\neg\texttt{hyp}(2) : - \ & \textbf{not } \texttt{terminated}(\texttt{perm}(\texttt{shoot}(\texttt{A1}, \texttt{A2})), \texttt{Time}), \\
& \texttt{holdsat}(\texttt{peace}, \texttt{Time}).
\end{aligned}$$

   Similarly, the revised rule in the second suggestion is captured by the first rule above with head predicate $\neg\texttt{hyp}(2)$, while the second of these represents the rule deleted by the third suggestion.

3. Remove: $\texttt{terminated}(\texttt{perm}(\texttt{shoot}(\texttt{A1}, \texttt{A2})), \texttt{Time}) : -\texttt{holdsat}(\texttt{peace}, \texttt{Time})$. This results in the following rules been added to the normative specification:

$$\begin{aligned}
\neg\texttt{hyp}(3) : - \ & \textbf{not } \texttt{terminated}(\texttt{perm}(\texttt{shoot}(\texttt{A1}, \texttt{A2})), \texttt{Time}), \\
& \texttt{initiated}(\texttt{peace}, \texttt{Time}).
\end{aligned}$$

   The above rule, with head predicate $\neg\texttt{hyp}(3)$, corresponds to the rule revised by the second revision suggestion.

Abducing Relevant Literals. Let $\langle T, O\rangle$ be the use-case that was used to learn the set R of suggested revisions, N_{B} be the part of the normative specification that R leaves unchanged, N_{R} the rules in the specification that one or more suggestions in R revise, $C_{\text{H}}/2$ be the function that transform rules by suggested revisions as described in section 4.2, and let *HYP* be the set of hypotheses in $C_{\text{H}}(N_{\text{R}}, R)$. The relevant literals are

solutions of the abductive task $\langle B, Ab, G\rangle$ where:

$$B = N_{\mathrm{B}} \cup T \cup C_{\mathrm{H}}(N_{\mathrm{R}}, R)$$
$$G = O \cup \neg(\bigwedge_{H_i \in \boldsymbol{HYP}} \neg H_i) \cup \neg(\bigwedge_{H_i \in \boldsymbol{HYP}} H_i)$$

and Ab is the set of ground instances of (possible) outcomes. The relevant literals is a set $E \subseteq Ab$ such that $B \cup E \models G$.

The above abductive task is computed using ASP and the solutions generated are answer sets containing relevant literals. To know the exact impact each relevant literal has on the hypothesis space, it is important to match it to the hypotheses it refutes. Algorithm 1 is used to extract relevant literals that refute a given suggested change (i.e. learned hypothesis) from the answer sets, using a series of set comparisons. The algorithm finds the differences between an answer set with a falsified hypothesis and another where it is not, then finds the smallest subsets of all these differences. The output of the algorithm are the smallest sets of literals that can refute the hypothesis. Note that while set intersection could potentially be used to extract such relevant literals, it would disregard the cases where a disjunction of literals $l_1 \vee l_2$ can falsify a hypothesis.

Scoring Relevant Literals. Ideally we want to be able to dismiss as many suggested revisions as possible. The number of hypotheses that could be discarded depends on the relevant literal's truth value: e.g. while we may be able to reject nearly all hypotheses if the literal is true, we may not be able to reject any should it be false. We use the number of minimum hypotheses that a relevant literal may reject as the score for comparing the literal against other relevant literals, using a fractional score when the literal can only falsify a hypothesis in conjunction with others. Thus, for each relevant literal l that rejects n suggested revisions when it is true, and m suggested revisions when it is false, $minimum(n, m)$ is the score for l. The most relevant literals are those with the highest value of these scores, and could be further ranked according to the maximum number of hypotheses each one falsifies.

5 Case Study

The case study is taken from [4]. The scenario describes a system of file sharing agents where:

> Agents are initialized to have ownership of a unique block of digital data, which all together comprise a digital object – a file of some kind. After the initial download of the unique block, an agent must share a copy of a block of data it possesses before acquiring the right to download a copy of a block from another agent. Violations and misuses are generated when an agent requests a download without having shared a copy of a block after its previous download, and a misuse terminates its empowerment to download blocks. However, if an agent has *VIP* status, it can download blocks without any restriction.

Algorithm 1 Extracting relevant literals of a given hypothesis

Input: answer sets ANS, hypothesis predicate h, and the set of hypothesis predicates HYP
Output: a set
REV of relevant literals that refute h

1: {Find the difference between S_i and answer sets that do not have relevant literals of h}
2: $DIFF = \emptyset$
3:
4: **for all** $S_i \in ANS$ **do**
5: **if** $\neg h \in S_i$ **then**
6: **for all** $S_j \in ANS$ **do**
7: **if** $\neg h \notin S_j$ **then**
8: $NREV = S_j \cup HYP \cup \{\neg h : HYP\}$
9: $DIFF = DIFF \cup \{S_i - NREV\}$
10: **end if**
11: **end for**
12: **end if**
13: **end for**
14:
15: {Find the smallest subsets from the sets in $DIFF$}
16: $REV = \emptyset$
17:
18: **for all** $D \in DIFF$ **do**
19: $REV = REV - \{R : REV | R \supset D\}$
20: **if** $D \notin REV$ and $\nexists R : REV(R \subset D)$ **then**
21: $REV = REV \cup \{D\}$
22: **end if**
23: **end for**
24:
25: **return** REV

Our existing normative specification includes the six revisable rules in Fig.3(a), that is N_{T}. The learner is supplied with the use-case comprising T (Fig.3(b)) and O (Fig.3(c)). This use-case shows how a violation is raised when $alice$ downloads data consecutively without sharing any data in between. On the other hand, no violations are raised when $charlie$ downloads data without sharing, as $charlie$ is a VIP. For the system specification to comply with the use-case, the fourth and fifth rule need to be revised, so that VIP agent's empowerment will not be terminated after a download, and a syntactic error in the fifth rule corrected, where the first `Y` should be `X` in `occurred(download(Y,Y,B),I).`

For this particular use-case and six revisable rules, with a maximum of seven rules per solution the learner outputs 41 ways in which the rules could be revised. Due to the space limitations, we look only at 4 of the proposed 41 (see Fig. 4).

Fig. 3. Rules for revision (a), with use-case trace (b) and outputs (c)

5.1 Generating Relevant Literals

To form the background knowledge for the abductive task, `rule 4` and `rule 5` are removed from the current specification, and their suggested revisions included in the specifications following the representation described in section 4.2. Fig. 5 contains an extract from the ASP encoding of our abductive task for computing relevant literals regarding revisions for `rule 4` and `rule 5`.

By adding the trace, as well as these hypotheses to the framework, the program can be used as the background data for the abduction task. The head of the suggested new and revised rules are used as abducible predicate symbols, while their revised conditions are used as constraints for these abducibles to avoid an explosion in the number of answer sets. The following integrity constraints capture conditions 2 and 4 of our test characterisation given in section 4.2

```
:- hyp(1), hyp(2), hyp(3), hyp(4).
:- -hyp(1), -hyp(2), -hyp(3), -hyp(4).
```

However, since we use Algorithm 1 to identify the relevant literals, as explained in section 4.2 the constraint is relaxed to:

```
%---Suggestion 1
% New rule
occurred(misuse(A),I) :- occurred(viol(myDownload(A,C)),I).
% Revise rule 4
terminated(perm(myDownload(X,B2)),I) :- occurred(myDownload(X,B),I), not isVIP(X).
% Revise rule 5
occurred(myDownload(X,B),I) :-
   holdsat(hasblck(Y,B),I), occurred(download(X,Y,B),I).

%---Suggestion 2
% New rule
occurred(misuse(A),I) :- occurred(viol(myDownload(A,C)),I).
% Revise rule 4
terminated(perm(myDownload(X,B2)),I) :- occurred(myDownload(X,B),I), not isVIP(X).
% Revise rule 5
occurred(myDownload(X,B),I) :-
   holdsat(hasblck(Y,B),I), occurred(download(X,Y,B),I).
occurred(myDownload(X,B),I) :-
   holdsat(hasblck(Y,B),I), occurred(viol(myDownload(Y,B2)),I).

%---Suggestion 3
% New rule
occurred(misuse(A),I) :- occurred(viol(myDownload(A,C)),I).
% Revise rule 4
terminated(perm(myDownload(X,B2)),I) :- occurred(myDownload(X,B),I), not isVIP(X).
% Revise rule 5
occurred(myDownload(X,B),I) :- occurred(download(X,Y,B),I).

%---Suggestion 4
% New rule
occurred(misuse(A),I) :- occurred(viol(myDownload(A,C)),I).
% Revise rule 4
terminated(perm(myDownload(X,B2)),I) :- occurred(myDownload(X,B),I), not isVIP(X).
% Revise rule 5
occurred(myDownload(X,B),I) :-
   holdsat(hasblck(Y,B),I), occurred(download(X,Y,B),I).
occurred(myDownload(X,B),I) :-
   holdsat(hasblck(Y,B),I), occurred(viol(myDownload(X,B2)),I).
```

Fig. 4. 4 selected revision suggestions from the 41 proposed

```
:- -hyp(1), -hyp(2), -hyp(3), -hyp(4).
```

The constraint above is still needed, as the algorithm searches for answer sets which includes `-hyp/1` instances to extract relevant literals from. Thus, while the answer sets without any refuted hypothesis are excluded from the algorithm's output, answer sets with all hypotheses refuted will still be included.

Applying Algorithm 1 to the answer sets generated by the abductive task, the following relevant literals are computed:

Literals that can falsify both $hyp(2)$ and $hyp(4)$:

```
¬ occurred(viol(myDownload(alice,x1)),i06),
¬ occurred(viol(myDownload(alice,x2)),i06),
¬ occurred(viol(myDownload(alice,x3)),i06),
¬ occurred(viol(myDownload(alice,x5)),i06),
¬ occurred(viol(myDownload(bob,x1)),i06),
¬ occurred(viol(myDownload(bob,x2)),i06),
¬ occurred(viol(myDownload(bob,x3)),i06),
¬ occurred(viol(myDownload(bob,x4)),i06),
¬ occurred(viol(myDownload(bob,x5)),i06)
```

```
% New Rule
-hyp(H) :- not occurred(misuse(A),I), occurred(viol(myDownload(A,C)),I), hyp_id(H)
    .
% Rule 4
-hyp(H) :- not terminated(perm(myDownload(X,B2)),I), occurred(myDownload(X,B),I),
           not isVIP(X), hyp_id(H).

%---Suggestion 1
% Rule 5
-hyp(1) :- not occurred(myDownload(X,B),I), occurred(download(X,Y,B),I),
           holdsat(hasblck(Y,B),I).

%---Suggestion 2
% Rule 5
-hyp(2) :- not occurred(myDownload(X,B),I), occurred(download(X,Y,B),I),
           holdsat(hasblck(Y,B),I).
-hyp(2) :- not occurred(myDownload(X,B),I), occurred(viol(myDownload(Y,B2)),I),
           holdsat(hasblck(Y,B),I).

%---Suggestion 3
% Rule 5
-hyp(3) :- not occurred(myDownload(X,B),I), occurred(download(X,Y,B),I).

%---Suggestion 4
% Rule 5
-hyp(4) :- not occurred(myDownload(X,B),I), occurred(download(X,Y,B),I),
           holdsat(hasblck(Y,B),I).
-hyp(4) :- not occurred(myDownload(X,B),I), occurred(viol(myDownload(X,B2)),I),
           holdsat(hasblck(Y,B),I).
```

Fig. 5. Computing relevant literals

Literals that can falsify only $hyp(4)$:

```
occurred(misuse(bob),i06) ∧ occurred(viol(myDownload(bob,x1)),i06)
occurred(misuse(bob),i06) ∧ occurred(viol(myDownload(bob,x2)),i06)
occurred(misuse(bob),i06) ∧ occurred(viol(myDownload(bob,x3)),i06)
occurred(misuse(bob),i06) ∧ occurred(viol(myDownload(bob,x5)),i06)
```

However, $hyp(1)$ and $hyp(3)$ cannot be falsified as both revisions produce the same consequences using the current use-case.

5.2 Scoring the Relevant Literals

When scoring the literals such as `occurred(misuse(bob),i06)`, where the literal alone cannot refute a hypothesis, a fractional score is given corresponding to how many other literals are needed to reject the hypothesis. The scores for each relevant literal are given in Table 1, with the following four literals having highest score:

```
occurred(viol(myDownload(bob,x1)),i06)
occurred(viol(myDownload(bob,x2)),i06)
occurred(viol(myDownload(bob,x3)),i06)
occurred(viol(myDownload(bob,x4)),i06)
occurred(viol(myDownload(bob,x5)),i06)
```

Any of these literals can be returned to the designer as the most relevant. Should the designer consider the returned literal to be false, then both the second and fourth suggested revisions could be discarded. However, if the literal is considered to be true, the dependent literal `occurred(misuse(bob),i06)` is given to the designer. This is be-

Relevant literal	Truth value	
	True	False
occurred(viol(myDownload(alice,x1)),i06)	0.0	2.0
occurred(viol(myDownload(alice,x2)),i06)	0.0	2.0
occurred(viol(myDownload(alice,x3)),i06)	0.0	2.0
occurred(viol(myDownload(alice,x5)),i06)	0.0	2.0
occurred(viol(myDownload(bob,x1)),i06)	0.5	2.0
occurred(viol(myDownload(bob,x2)),i06)	0.5	2.0
occurred(viol(myDownload(bob,x3)),i06)	0.5	2.0
occurred(viol(myDownload(bob,x4)),i06)	0.5	2.0
occurred(viol(myDownload(bob,x5)),i06)	0.5	2.0
occurred(misuse(bob),i06)	0.0	0.5

Table 1. Scoring of relevant literals

cause the two literals are dependent as shown by the lists of relevant literals for each hypothesis given above.

6 Related work

The literature on norm change and norm revision is quite diverse, but also quite thinly spread across a range of disciplines. Many normative frameworks include appeal to extrinsic normative frameworks, such as negotiation, argumentation, voting, or even fiat (dictatorship) to mediate norm change. These are not the concern of this paper. Our focus is on the specific nature of the revision: *what* needs to change, rather than *how* it shall be brought about. In human societies, the identification of what may be informal, or the outcome of an extensive evaluative study, along with proposals for which rules to revoke, which rules to add and an assessment of the consequences. This reflects work in the philosophy of law, the logic of norms, or the logic of belief change [17], where the drive has been the discovery of norm conflicts and their subsequent revision in the framework of deontic logic. However, this only explores the principle of norm conflict and norm inconsistency (concluding that they are in fact the same), and that it may be resolved by a process of norm revision in which the norm set is reduced and subsequently extended (consistently). Further theoretical studies can be found in [18–20]

Artikis [21] presents a formalization of a (run-time) process for changing the rules governing a protocol, central to which are the notions of stratification and degrees of freedom to determine a metric for the magnitude and hence feasibility of the change from the current rules to the new rules. However, fundamental to this scheme is that the state-space of alternatives be known *a priori*, so it is essentially limited to known-knowns, rather than the exploration of all possibilities to remedy shortcomings.

Campos et al. [22] propose a mechanism for the adaptation of a normative framework – which they call an electronic institution (EI) – in which the EI is goal-driven and utilizes a feedback mechanism to compare observations with expected goals in order to self-reconfigure using transition functions. The expected goals are quantitative

constraints on values of observed properties, while actual performance is captured in an objective function comprising a weighted aggregation of observed properties. As with Artikis, above, the scope for adaptation is limited in that responses are pre-determined in the specification and may only affect parameters of norms.

Tinnemeier et al. [23] make clear that normative concepts should be used to affect which entities have the permission and the power to effect norm change, they also point out that norm-reasoning is typically beyond the competence of typical agency. In consequence of the latter, they choose for the normative framework to provide suitable norm-change operators for the agents to use that do not require detailed norm knowledge. While the scope of changes is more extensive than either Artikis or Campos, the rules for norm scheme change (sic) appear to depend both on domain knowledge and the foresight of the designer.

Thus, although there is a select literature which addresses norm change in various ways, it either suffers from an absence of a computational model, or has very restricted solution space which depends on prediction of what changes may be needed. In contrast, we provide technical support for a formal model of norm revision, as presented here and in our earlier paper [4], which can adapt the normative framework arbitrarily, to meet evolving requirements, expressed through goals, and is, we believe, entirely novel.

7 Conclusions

In this paper, we have tackled the problem of distinguishing between revisions of normative specifications through the use of test generation. While we have concentrated on problem of choosing between normative revisions, more generally our work for choosing between alternate hypotheses is applicable to any theory revision problem. As discussed in [11], there appears two ways of judging whether one revision is better than another. The first is by looking at how complete and consistent the revised theory would be by checking it satisfies a set of desired characteristics such as the AGM postulate (see Chapter 2 of [24]). As we depended on previous work for the correctness of the revision, this is not the directly related to our work. The second is by following the principle of minimal change which takes the revision that changes the original theory the least as the best solution. While this approach ensures that as much knowledge as possible is retained by the change, the minimal revision may not always reflect the specification that the designer wants. Thus, other criteria in addition to minimal changes should be used in a revision framework based on use-cases.

Although examining all possible revisions may give a more complete view of the changes made to the original partial specification, our approach can help the user by pointing out the key discriminating aspects between the different revisions. By identifying comparable consequences of the suggested revisions, we are able to use them as a rationale for rejecting possible changes. We have investigated how test generation can be applied, providing a notion of test characteristics for revisions, and used this characterisation to describe how abduction can be used to find such relevant literals. In [15], the *discriminating test* is mentioned as another type of test that could reject hypotheses regardless of its truth value. It is also mentioned that while they are ideal to use for rejecting hypotheses, their characteristics are too restrictive and thus relevant tests

were discussed. For the relevant literals in this paper, the scoring mechanism ensures that relevant literals satisfying the characteristics of discriminating tests have higher priority.

Our case study demonstrates how our proposed approach could be integrated into an existing framework for normative refinement, where ASP can be used to compute relevant literals and score them in order to identify those that are most relevant. It also shows a situation where our approach may not discriminate all suggested revisions. While the revision suggestions are different, our approach could not find any relevant literals as the system trace used to find it does not describe a scenario in which the revisions would differ. As the revision process is designed to be carried out iteratively, use-cases from previous cycles could be kept either as additional constraints or as additional traces to use for generating relevant literals.

References

1. Grossi, D., Aldewereld, H.M., Dignum, F.: Ubi lex, ibi poena: Designing norm enforcement in e-institutions. In Noriega, P., Vázquez-Salceda, J., Boella, G., Boissier, O., Dignum, M., Fornara, N., Matson, E., eds.: COIN II, Springer (2007) 101–114
2. Cliffe, O., De Vos, M., Padget, J.: Answer set programming for representing and reasoning about virtual institutions. In Inoue, K., Satoh, K., Toni, F., eds.: Computational Logic in Multi-Agent Systems. Volume 4371 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2007) 60–79
3. Artikis, A., Sergot, M., Pitt, J.: An executable specification of an argumentation protocol. In: Proceedings of Conference on Artificial Intelligence and Law (ICAIL), ACM Press (2003) 1–11
4. Corapi, D., Russo, A., Vos, M.D., Padget, J.A., Satoh, K.: Normative design using inductive learning. TPLP **11**(4-5) (2011) 783–799
5. Searle, J.R.: A Construction of Social Reality. Allen Lane, The Penguin Press (1955)
6. Jones, A.J., Sergot, M.: A Formal Characterisation of Institutionalised Power. ACM Computing Surveys **28**(4es) (1996) 121 Read 28/11/2004.
7. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing **9**(3-4) (1991) 365–386
8. Kowalski, R., Sergot, M.: A logic-based calculus of events. New Gen. Comput. **4**(1) (1986) 67–95
9. Gelfond, M., Lifschitz, V.: Action languages. Electron. Trans. Artif. Intell. **2** (1998) 193–210
10. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: Engineering an incremental ASP solver. In Garcia de la Banda, M., Pontelli, E., eds.: Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08). Volume 5366 of Lecture Notes in Computer Science., Springer-Verlag (2008) 190–205
11. Wrobel, S.: First order theory refinement (1996)
12. Sakama, C.: Induction from answer sets in nonmonotonic logic programs. ACM Trans. Comput. Log. **6**(2) (2005) 203–231
13. Corapi, D.: Nonmonotonic Inductive Logic Programming as Abductive Search. PhD thesis, Imperial College London (2012)
14. Corapi, D., Russo, A., Lupu, E.: Inductive logic programming in answer set programming. In: ILP. (2012) 91–97
15. Mcilraith, S.: Generating tests using abduction. In: Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning (KR'94), Morgan Kaufmann (1994) 449–460

16. Kakas, A.C., Kowalski, R., Toni, F.: Abductive logic programming. Journal of Logic and Computation 2(6) (1992) 719–770
17. Alchourrón, C.E.: Conflicts of norms and the revision of normative systems. Law and Philosophy **10** (1991) 413–425 10.1007/BF00127412.
18. Ullmann-Margalit, E.: Revision of norms. Ethics **100**(4) (July 1990) 756–767 Article Stable URL: http://www.jstor.org/stable/2381777, retrieved 20120320.
19. Boella, G., van der Torre, L.W.N.: Regulative and constitutive norms in normative multiagent systems. In Dubois, D., Welty, C.A., Williams, M.A., eds.: KR, AAAI Press (2004) 255–266
20. Governatori, G., Rotolo, A.: Changing legal systems: legal abrogations and annulments in defeasible logic. Logic Journal of the IGPL **18**(1) (2010) 157–194
21. Artikis, A.: Dynamic protocols for open agent systems. In Sierra, C., Castelfranchi, C., Decker, K.S., Sichman, J.S., eds.: AAMAS (1), IFAAMAS (2009) 97–104
22. Campos, J., López-Sánchez, M., Rodríguez-Aguilar, J.A., Esteva, M.: Formalising situatedness and adaptation in electronic institutions. In Hübner, J.F., Matson, E.T., Boissier, O., Dignum, V., eds.: COIN@AAMAS&AAAI. Volume 5428 of Lecture Notes in Computer Science., Springer (2008) 126–139
23. Tinnemeier, N.A.M., Dastani, M., Meyer, J.J.C.: Programming norm change. In van der Hoek, W., Kaminka, G.A., Lespérance, Y., Luck, M., Sen, S., eds.: AAMAS, IFAAMAS (2010) 957–964
24. Gabbay, D.M., Rodrigues, O., Russo, A.: Revision, Acceptability and Context - Theoretical and Algorithmic Aspects. Cognitive Technologies. Springer (2010)