Programming Norm Change

Nick Tinnemeier
Utrecht University
The Netherlands
nick@cs.uu.nl

Mehdi Dastani
Utrecht University
The Netherlands
mehdi@cs.uu.nl

John-Jules Meyer
Utrecht University
The Netherlands
jj@cs.uu.nl

ABSTRACT

To adequately deal with the unpredictable and dynamic environments normative frameworks are typically deployed in, mechanisms for modifying the norms at runtime are crucial. We present the syntax and operational semantics of generic programming constructs to facilitate runtime norm modification, allowing a programmer to specify when and how the norms may be changed by external agents or by the normative framework. The norms take on the form of conditional obligations and prohibitions, which instantiate detached obligations and prohibitions (instances). We present rule-based constructs for runtime modification of the norms and their instances, and a mechanism for automatically updating the instances when their underlying norms change.

Categories and Subject Descriptors

I.2 [**Artificial Intelligence**]: Programming Languages and Software; D.3 [**Programming Languages**]: Miscellaneous

General Terms

Languages, Design, Theory

Keywords

Dynamic Norms, Programming, Operational Semantics

1. INTRODUCTION

Multi-agent systems are a potentially powerful means to build complex software systems in which autonomous, heterogeneous and independently designed agents may dynamically enter and exit. Because typically little can be assumed about the behavior individual agents will exhibit, regulating their behavior has become a major challenge. The use of organizational abstractions in general [16], and norms in particular [14], have been widely promoted as an approach for coordinating individual agents. The idea is that a multi-agent system is equipped with a set of norms that dictate the ideal behavior the agents ought to exhibit. Indeed, practical approaches that employ a normative framework – a computational entity that is responsible for checking for violations and fulfillments of the norms and applying sanctions accordingly – in the construction of multi-agent systems are rapidly emerging (see [9] for some examples).

Cite as: Programming Norm Change, N.A.M. Tinnemeier, M. Dastani and J-J. Ch. Meyer, *Proc. of 9th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2010)*, van der Hoek, Kaminka, Lespérance, Luck and Sen (eds.), May, 10–14, 2010, Toronto, Canada, pp. 957-964

Facing the unpredictable and dynamic nature of the environments normative frameworks are deployed in a static view in which the norms are specified at design time and cannot be modified at runtime does often not suffice [7, 4, 10]. Modifying norms at runtime increases the system's flexibility to manage a priori unforeseen situations. Consider a conference management system example. If many reviewers indicate that they will not be able to comply with the norm specifying that reviews ought to be uploaded before a certain deadline, the system might react to this by relaxing this norm. Although a considerable amount of work has been devoted to the theoretical aspects of norm change (examples are [1, 3, 11]), rather less attention has been paid to the practical aspects related to computational mechanisms of norm change. Similar to [2, 5, 6] the focus of this paper is on a computational mechanism for runtime norm change.

Norms are typically specified as conditional sentences defining under which circumstances deontic concepts such as obligations, permissions and prohibitions are established. If, for instance, a reviewer is assigned a paper, an obligation to have uploaded its review for that paper is created. A key question is then what happens to the obligations, permissions and prohibitions when the underlying norms change. This issue has been theoretically investigated in the context of modifying legal systems [11]. However, existing work [2, 5, 6] on computational norm change considers changing the normative specification, but does not address the intricacies of how the associated deontic concepts may evolve.

Another issue related to norm change pertains to who is responsible for deciding under which circumstances and how the norms are changed. Some (e.g. [7, 2]) consider it the task of the agents to alter the norms. Yet others (e.g. [5, 6]) consider it the responsibility of the normative framework to autonomously modify the norms. However, none of the aforementioned practical frameworks of norm change is general purpose in the sense that it unifies both approaches.

This paper contributes to the operationalization of runtime norm change, in particular:

- We describe the overall architecture of a norm change mechanism within the context of a multi-agent system regulated by norms in section 2. Contrasting existing work on the operationalization of norm change, the solution we develop enables external agents as well as the normative framework to change the norms.
- We present generic programming constructs (section 4) for the runtime modification of norms and their deontic in-

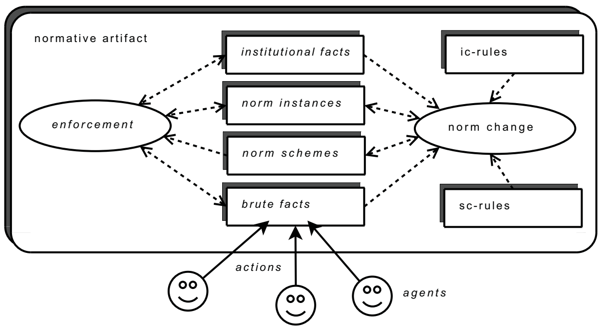

Figure 1: A normative artifact. Arrows from store (rectangle) to process (ellipse) denote modification and arrows in the opposite direction denote reading of the store's elements.

stances, also focusing on a construct for automatically updating the deontic instances when their underlying norms change. This has not been done before in respect of a computational norm change mechanism.

- We formalize the programming constructs with an operational semantics [12] (section 5). This allows us to study the proposed constructs in a mathematically rigorous way and is already close to the implementation of an interpreter. To our best knowledge, we are the first to study the operational semantics of norm change.

The norms that are subject to change are taken from [15] and are briefly explained by means of an example of a conference management system in section 3. This example will be used to illustrate and motivate our architecture throughout the rest of the paper. We compare our work with related work in section 6 and conclude in section 7.

2. NORM ENFORCEMENT AND CHANGE

Norm change cannot be considered without the representation of norms and the mechanism responsible for enforcing them. As a starting point we use the norms and accompanying enforcement mechanism presented in [15]. The choice for this work is mainly motivated by the declarative nature of these norms and the presence of an operational semantics, facilitating the definition of the operational semantics of the norm change mechanism.

We conceive a multi-agent system regulated by norms as a set of agents interacting with one (or more) normative artifact(s). Inspired by the agents and artifacts approach [13], a normative artifact encapsulates a domain specific state and function that agents may use to achieve their objectives. The conceptual architecture is shown in figure 1 with the concepts originally introduced in [15] shown in italics.

The domain specific state is modeled by the *brute facts* which can be used, for example, to store information about uploaded papers and their reviews. Agents may exploit the function provided by the artifact by performing *actions*, which manipulate the brute facts. The rules of conduct that help protect and maintain the design objectives of the artifact and guide the agents in interacting with it in a meaningful way are specified by the *norm schemes*. They are conditional obligations and prohibitions that instantiate *norm instances* (detached obligations and prohibitions) when their condition is satisfied. Norm instances describe which brute states should be achieved or avoided. The *enforcement mechanism* is responsible for creating norm instances, detecting their fulfillment or violation, and punishing agent's wrongdoings by imposing sanctions on the brute state. Information about (amongst others) norm violation and which roles agents have enacted is stored by the *institutional facts*.

To allow for norms to change at runtime we enrich the normative artifact with a separate *norm change mechanism.* Our mechanism and its architecture is motivated by two key principles. That is, norm change is:

*system-dependent***:** how, who and under which circumstances norms and their instances may be changed must be specified by the normative framework;

*enforcement-independent***:** the norm change mechanism must be defined separately from the enforcement mechanism.

The first principle accords with the definition of normative multi-agent system presented in [4] which includes that *"the normative systems specify how and in which extent the agents can modify the norms."* Clearly, if the system does not put restrictions on who and to which extent norms can be changed the power of the system to regulate the agents' behavior would be compromised. Moreover, deciding when and how norms are changed requires knowledge about and capabilities to reason with norms, fulfillments and violations. The normative framework might benefit from keeping this information private. Most importantly, norm-reasoning capabilities are beyond the competences of typical agency and requiring such a specialized capability limits the types of agents that can interact with the framework, thereby restricting its openness. Therefore, we argue that the normative framework should provide suitable actions by which the agents can change the norms without needing detailed knowledge about their structure. This accords with the principle of encapsulation (a.k.a. data hiding).

The second principle states that the norm enforcement mechanism should be defined independently from the norm change mechanism. This implies that specifying the code that is concerned with norm change must be explicitly defined as a separate component rather than entangling it with the code defining the norm schemes. This pertains to a separation of concerns, promoting readability and manageability of the program. Another advantage is that (if desired) different computation mechanisms could be used for the norm change mechanism without affecting the norm enforcement mechanism and vice versa. In principle, the program that defines how the norms may change could even be plugged in at runtime to deal with unforeseen situations even more flexibly. This is, however, beyond the scope of this paper.

The norm change mechanism is driven by two kinds of constructs that are defined by the programmer. Both types are rules consisting of a precondition that describes under which circumstances the rule is applicable and a consequent that specifies the changes to be made. The norm schemes can be altered through the norm *scheme change rules*, which will be abbreviated by sc-rules. The norm instances pertaining to the active obligations and prohibitions can be changed through the norm *instance change rules*, ic-rules for short.

The preconditions of the rules can range over brute facts, institutional facts and norm instances that are in effect. This way, one the one hand, the rules can be designed to assess the situation of the artifact at runtime and express how the

⟨id⟩	a label uniquely identifying a norm scheme.
⟨role⟩	a symbol identifying a role that can be played.
⟨b-atom⟩	a first-order atom denoting a brute fact.
⟨i-atom⟩	a first-order atom denoting an institutional fact.
⟨r-atom⟩	a first-order atom of the form $rea(i, r)$ in which r denotes a role and i the agent playing it.
⟨action⟩	a first-order atom of the form $P(i, t_1, \ldots, t_n)$ in which predicate P denotes the name of the action, i the agent performing it, and $t_1, \ldots, t_n$ the action's arguments.

Table 1: Elementary syntactical constructs.

```
⟨artifact⟩ = "Roles:" ⟨role⟩ {⟨role⟩}
             "Facts:" ⟨b-lit⟩ {⟨b-lit⟩}
             "Effects:" ⟨effect⟩ {⟨effect⟩}
             "Norms:" ⟨norm⟩ {⟨norm⟩};
⟨effect⟩   = "{" ⟨pre⟩ "}" ⟨action⟩ "{" ⟨post⟩ "}";
⟨pre⟩      = ⟨b-lit⟩ | ⟨r-lit⟩ | ⟨pre⟩ { "," ⟨pre⟩ };
⟨post⟩     = ⟨b-lit⟩ | ⟨post⟩ { "," ⟨post⟩ };
⟨norm⟩     = ⟨id⟩ ":<" ⟨cond⟩ "," ⟨OP⟩ "," ⟨ddln⟩ ">";
⟨cond⟩     = ⟨b-lit⟩ | ⟨r-lit⟩ | ⟨cond⟩ "and" ⟨cond⟩;
⟨OP⟩       = "O(" ⟨b-test⟩ ")" | "F(" ⟨b-test⟩ ")";
⟨ddln⟩     = ⟨b-test⟩;
⟨b-lit⟩    = ⟨b-atom⟩ | "not" ⟨b-atom⟩;
⟨r-lit⟩    = ⟨r-atom⟩ | "not" ⟨r-atom⟩;
⟨i-lit⟩    = ⟨i-atom⟩ | "not" ⟨i-atom⟩;
⟨b-test⟩   = ⟨b-lit⟩ | ⟨b-test⟩ "and" ⟨b-test⟩;
```

Figure 2: EBNF grammar of normative artifacts.

artifact could autonomously change the norms accordingly. On the other hand, because agents can alter the brute facts by the execution of actions, the norm change rules can also be designed to empower agents to change the norms.

It is important to emphasize that the issue of how the enforcement and change mechanism are scheduled is beyond the scope of this paper. The triggering, fulfillment or violation of a norm might enable the application of a norm change rule, whereas a change to the norms again might necessitate enforcement of these modified norms. Norms will most likely not change frequently and balancing between enforcement and change with the goal of increasing computational efficiency is also application specific.

3. PROGRAMMING WITH NORMS

In this section we briefly discuss the syntax and intuitive semantics of the programming language presented in [15] by which normative artifacts can be programmed, and in particular, the norms that are subject to change. It should be emphasized that the norms presented here are a simplified version of the ones in [15]. We do not consider norms that specify obligations and prohibitions that arise in the sub-ideal situation an agent does not abide by the norms. Nor do we include sanctions in the norm schemes. Sanctioning could be considered the agents' responsibility or a separate sanctioning mechanism as, for instance, can be found in [8] could be used. There is no particular difficulty for the norm change mechanism in including sanctions in the norm schemes. Also, we omit the parameters of the unique labels by which norm schemes are identified. These parameters are typically used for storing information such as the debtor of a norm. We limit the representation to promote generality and simplicity. The focus is on norm change, it is not our goal to extend the expressiveness of the norms of [15].

The syntax by which artifacts can be programmed is depicted in figure 2 and the basic syntactical elements are explained in table 1. We briefly explain these constructs in more detail by means of an example involving a conference management system guiding agents playing the role of author, reviewer and chair. The system can be in different phases, namely the phase in which the system is closed, abstracts can be uploaded, papers can be uploaded, papers are reviewed, reviews are collected and authors are notified. The code of this program is partially listed in code fragment 3.1. To program this system is to specify the roles that can be played (line 1) and the initial brute state (line 2). The fact `id(X)` is used to remember the identifier of the last unique paper id that has been assigned. The effect rules specify how the brute facts evolve under the performance of actions by agents. They are of the form $\{\Phi\}\alpha\{\Psi\}$, intuitively meaning that if action α is executed and the facts Φ hold in the current brute state, the facts Ψ will be accommodated to the new brute state. Lines 3-10 specify the actions for opening the system; uploading abstracts, papers and reviews; and (re)assigning papers to reviewers.

The desired behavior is dictated by the norm schemes taking on the form of conditional obligations and prohibitions, expressed as tuples of the form $l : \langle \varphi_c, \mathbb{P}(\varphi_x), \varphi_d \rangle$ with $\mathbb{P}$ either an obligation denoted by O or a prohibition denoted by F. The intuitive reading of a conditional obligation is that "if condition φ_c holds then there is an obligation to establish the brute state denoted by φ_x before deadline φ_d is satisfied by the brute state." A conditional prohibition can be intuitively read as "if condition φ_c holds it is forbidden to establish the brute state denoted by φ_x before φ_d."

The norm schemes for the conference management system are listed on lines 10-14. The first norm scheme expresses that uploaded papers should not exceed the page limit of 15 pages[1]. Suppose an author, say `jane`, has uploaded an abstract and has been assigned id `547`. As soon as the chair puts the system in the submission phase the condition is satisfied and the norm scheme is instantiated in a norm instance `(pagelimit, F(pages(547)>15), phase(review))` stating that `jane`'s paper is not allowed to exceed 15 pages. It stays into effect during the whole submission phase, i.e. until the review phase starts. A violation is detected as soon as `jane` uploads a paper of more than 15 pages. The second norm scheme states that a reviewer is obliged to have uploaded its assigned reviews before the reviews will be collected. This obligation becomes active as soon as a reviewer is assigned a paper and stays into effect until it is either fulfilled (the review has been uploaded) or it is violated (the review has not been uploaded before the deadline). The final norm specifies that by the end of the reviewing phase there should be at least three reviews per paper. Note that the debtor (the chair) is not explicitly stated by this norm scheme as a consequence of the aforementioned omission of the label's parameters.

4. PROGRAMMING NORM CHANGE

We introduce two types of norm change rules, viz. norm instance change rules (ic-rules for short) and norm scheme change rules (sc-rules for short). More concretely, we extend the grammar of a normative artifact with:

```
"IC-rules:" ⟨ic-rule⟩ { ⟨ic-rule⟩ }
"SC-rules:" ⟨sc-rule⟩ { ⟨sc-rule⟩ }
```

The grammar of ic-rules and sc-rules is shown in figure 3. Both types of rules will be explained in more detail below.

[1]To improve readability we introduce comprehensive names, such as `pages(PId) > 15`, for the atoms which clarify their meaning.

Code fragment 3.1 Conference management system.

```
Roles: chair, author, reviewer
Facts: phase(closed), id(0)
Effects:
{rea(C,chair), phase(closed)} open(C)   {not phase(closed), phase(abstracts)}
  :
{rea(A,author), phase(abstracts), id(PId)} uploadAbstract(A) {abstract(A,PId), not id(PId), id(PId+1)}
{rea(A,author), phase(submission), abstract(A,PId)} uploadPaper(A,PId) {paper(A,PId)}
{rea(C,chair), phase(review), paper(A,PId)} assign(C,R,PId) {assigned(R,PId)}
{rea(C,chair), assigned(R1,PId)} reassign(C,R1,PId,R2) {not assigned(R1,PId), reassigned(R1,PId,R2)}
{rea(R,reviewer), phase(review), assigned(R,PId)} uploadReview(R,PId) {review(R,PId)}
Norms:
pagelimit     : ⟨phase(submission) and abstract(A,PId), F(pages(PId) > 15), phase(review)⟩
review-due    : ⟨phase(review) and assigned(R,PId), O(review(R,PId)), phase(collect)⟩
min-reviews   : ⟨phase(submission) and paper(PId), O(nrReviews(PId) ≥ 3), phase(collect)⟩
```

```
⟨ic-rule⟩   = ⟨ant⟩ "=>" ⟨ic-cons⟩;
⟨ant⟩       = ⟨b-lit⟩ | ⟨i-lit⟩ | ⟨r-lit⟩ | ⟨ni⟩ | ⟨ant⟩ "and" ⟨ant⟩;
⟨ni⟩        = "(" ⟨id⟩ "," ⟨OP⟩ "," ⟨ddln⟩ ")";
⟨ic-cons⟩   = ⟨ic-list⟩ ⟨ic-list⟩;
⟨ic-list⟩   = "[" ⟨ni⟩ { "," ⟨ni⟩ } "]" | "[ ]";
⟨sc-rule⟩   = ⟨ant⟩ "=>" ⟨sc-cons⟩ | ⟨ant⟩ "=>*" ⟨sc-cons*⟩;
⟨sc-cons⟩   = ⟨sc-list⟩ ⟨sc-list⟩;
⟨sc-cons*⟩  = ⟨sc-list⟩ ⟨sc-list*⟩;
⟨sc-list⟩   = "[" ⟨norm⟩ { "," ⟨norm⟩ } "]" | "[ ]";
⟨sc-list*⟩  = "[" ⟨norm*⟩ { "," ⟨norm*⟩ } "]" | "[ ]";
⟨norm*⟩     = ⟨norm⟩ | "nil";
```

Figure 3: EBNF grammar of norm change rules.

4.1 Changing Norm Instances

The ic-rules offer a fine-grained mechanism to change one or more norm instances without changing their underlying norm schemes. They specify under which conditions and how changes to the norm instances are made. They are expressed as rules of the form $\beta \Rightarrow [ni_0, ..., ni_n][ni'_0, ..., ni'_m]$ with the intuitive reading that under circumstances β norm instances $ni_0, ..., ni_n$ are to be retracted and norm instances $ni'_0, ..., ni'_m$ are to be asserted. The rule's precondition ranging over brute facts, institutional facts and norm instances thus describes when the norm instances should be modified. How they should be modified is specified by the consequent.

Examples of these ic-rules in the context of our running example are given in code fragment 4.1 lines 1-5. Suppose that a reviewer `R1` informs the chair that he will not be able to fulfill his obligation to review a paper, then the chair might decide to reassign this paper to another reviewer, say `R2`. In this case the obligation for reviewer `R1` is transferred to reviewer `R2`, which boils down to removing the obligation of `R1` and creating a new one for `R2`. To give the new reviewer enough time to fulfill its obligation the deadline will be set to the notification phase instead of the collect phase. This is expressed by the first ic-rule. Recall the action `reassign(R1,PId,R2)` listed in code fragment 3.1 by which the chair can reassign paper `PId` from reviewer `R1` to `R2`, modifying the brute state such that the fact `assigned(R1,PId)` is retracted and a fact `reassigned(R1,PId,R2)` is asserted.

The first ic-rule is an example of a norm modification that is initiated by an agent. To illustrate a change of the norms on the decision of the normative framework consider the second ic-rule. Recall the norm of code fragment 4.1 that specifies that at the start of the collect phase at least three reviews should be uploaded for each paper. Assume that this implies that given the actual amount of uploaded papers and reviewers each reviewer would be assigned more than five papers. Suppose that the system reacts to this observation by relaxing the requirement of three reviews per paper by only requiring two. This is expressed by the second ic-rule that will modify all `min-reviews`' instances. Note that the variables of the ic-rule are thus implicitly universally quantified in the widest scope. Because (for the sake of the example) `min-reviews` only instantiates obligations in the submission phase there is no need for modifying the norm scheme also.

4.2 Changing Norm Schemes

Whereas ic-rules are used for altering the norm instances, sc-rules are used for modifying the norm schemes. The sc-rules take on a similar form as the ic-rules. More specifically, they are rules of the form $\beta \Rightarrow [ns_i, ..., ns_n][ns'_0, ..., ns'_m]$ in which β is a precondition that specifies when the norm schemes should be changed. How the norm schemes should be changed is specified by the two lists of the rule's consequent. The first list contains the norm schemes that are to be removed, whereas the second list contains the norm schemes that are to be added.

A key question in modifying a norm scheme is what happens to the norm instances it has already instantiated. In some cases it is desirable that the instantiated norm instances remain unaffected, whereas in other cases the associated norm instances should be changed accordingly. Consider, for example, a norm scheme that specifies that conference registrants are obliged to have paid a fee a week before the conference starts. Suppose that for some reason (the costs were higher than expected) we decide to increase this fee, then this increased fee will sensibly only apply for new registrants. In other words, we want the payment obligations that were in effect before the change to remain unaffected. If, however, the payment deadline is extended we might decide to apply this change retroactively. That is to say, the deadline of already existing payment obligations will also be extended. For this reason, we propose two types of norm scheme change rules. One that leaves the instantiated norm instances unaltered when their underlying norm scheme is changed and one that revises the associated norm instances accordingly. We name the first type of norm scheme change *instance-preserving* and the latter *instance-revising*. To distinguish between the two rules, the arrow of the instance-revising rules will be annotated with an asterisk, i.e. will take on the form $\Rightarrow_*$.

Updating a norm scheme is to delete the original norm scheme, say ns, and replacing it by a new one, say ns'. If an instance-preserving update is performed, all the instances of ns will remain intact. If, however, an instance-revising

Code fragment 4.1 Examples of norm change.

```
IC−rules:                                                                                       1
reassigned(R1,PId,R2) and (review−due, O(review(R1,PId), phase(P)) ⇒                            2
  [(review−due,O(review(R1,PId)),phase(collect))][(review−due,O(review(R2,PId)),phase(notify))]  3
phase(review) and 3∗nrPapers()/nrReviewers()>5 and (min−reviews, O(nrReviews(PId)≥3), phase(P)) ⇒   4
  [(min−reviews, O(nrReviews(PId)≥3), phase(P))][(min−reviews, O(nrReviews(PId)≥2), phase(P))]   5
SC−rules:                                                                                       6
nrPapers() > 10 and nrViolations(pagelimit)/nrPapers() > 0.25 ⇒*                                7
  [pagelimit:⟨phase(submission) and abstract(A,PId), F(pages(PId) > 15), phase(review)⟩,         8
   pagelimit:⟨phase(submission) and abstract(A,PId), F(pages(PId) > 15), phase(review)⟩]         9
  [pagelimit1:⟨phase(submission) and abstract(A,PId), F(15 < pages(PId) ≤ 17), phase(review)⟩,   10
   pagelimit2:⟨phase(submission) and abstract(A,PId), F(pages(PId) > 17), phase(review)⟩]        11
```

update is performed this means that each instance of ns is removed and transformed into an instance of ns'. We thus need to know by which norm scheme a scheme should be replaced. We relate a norm scheme with the norm scheme it should transform into by the position they respectively have in the retract and assert list, visualized:

$$\begin{array}{lll} \beta \Rightarrow_* & [& ns_0, ns_1, ..., ns_n \ ] \\ & & \ \downarrow \quad \downarrow \qquad \ \ \downarrow \text{ replaced by} \\ & [& ns'_0, ns'_1, ..., ns'_n \ ] \end{array}$$

As demonstrated by the following example, the transformation of norm schemes is not necessarily one on one. In fact, a norm scheme might evolve into multiple norm schemes and multiple norm schemes might be merged into one. If one only wants to remove a norm scheme ns together with all its associated instances, special element `nil` can be used to transform ns into an empty norm scheme.

Suppose that it is observed that the norm scheme `pagelimit` of code fragment 3.1 is often violated, e.g. more than ten papers have been uploaded of which more than 25% violates the page limit norm.[2] In reaction we could, for example, decide to allow authors to pay for an additional two pages and reject papers which exceed the limit by more than two pages. To be able to distinguish between violations that stay within the boundary of two pages and violations by more than two pages, we replace the `pagelimit` norm scheme by two norm schemes `pagelimit1` and `pagelimit2` as listed on lines 7–11 of code fragment 4.1. Because the `pagelimit` norm scheme evolves into two norm schemes it occurs twice in the removal list. To illustrate how the norm instances of norm scheme `pagelimit` are transformed by the application of this sc-rule consider the norm instance:

$$(\texttt{pagelimit}, \texttt{F}(\texttt{pages}(\texttt{547}) > \texttt{15}), \texttt{phase}(\texttt{review})) \qquad (1)$$

that was instantiated out of norm scheme `pagelimit` using substitution {`PId/547`}. To transform norm instance (1) into norm instances of `pagelimit1` and `pagelimit2` we use this same substitution. That is to say, we apply the substitution {`PId/547`} on the norm schemes `pagelimit1` and `pagelimit2`. After application of the sc-rule instance (1) will thus be transformed into the two norm instances:

$$(\texttt{pagelimit1}, \texttt{F}(\texttt{15} < \texttt{pages}(\texttt{547}) \leq \texttt{17}), \texttt{phase}(\texttt{review})) \qquad (2)$$

$$(\texttt{pagelimit2}, \texttt{F}(\texttt{pages}(\texttt{547}) > \texttt{17}), \texttt{phase}(\texttt{review})) \qquad (3)$$

[2] Even though we assume this information to be available, in [15] no history about violations is recorded. Extending the enforcement mechanism to store this information is beyond the scope of this paper.

5. OPERATIONAL SEMANTICS

An operational semantics describes the behavior of a programming language in terms of transitions between program configurations. A configuration describes a state of the program and a transition is a transformation of one configuration γ into another γ', denoted by $\gamma \rightarrow \gamma'$. Transitions are derived by derivation rules of the form $\frac{P}{\gamma \rightarrow \gamma'}$ with the intuitive reading that $\gamma \rightarrow \gamma'$ can be derived if premise P holds. An execution trace is then a sequence of transitions $\gamma_0 \rightarrow \gamma_1 \rightarrow \ldots \rightarrow \gamma_n$ (often written as $\gamma_0 \rightarrow^* \gamma_n$) derived by applying transition rules to an initial configuration.

As explained before, a norm instance is instantiated from its norm scheme when its condition is derivable from the brute and institutional state for some substitution of its formal parameters. Instantiating a norm scheme is to apply this substitution on it resulting in a norm instance. Henceforth, to denote the set of all variables $\overline{v}$ that occur in an unground norm instance or formula ϕ we write $\phi(\overline{v})$.

Definition 1 (Norm Instantiation). *Given a norm scheme $ns = l : \langle \varphi_c(\overline{v_1}), \mathbb{P}(\varphi_x(\overline{v_2})), \varphi_d(\overline{v_3}) \rangle$ in which $\overline{v_2} \cup \overline{v_3} \subseteq \overline{v_1}$ and a ground substitution θ for the variables the function inst that instantiates a norm instance from ns is defined as: $inst(ns, \theta) = (l, \mathbb{P}(\varphi_x(\overline{v_2})), \varphi_d(\overline{v_3}))\theta$*

Note the restriction on the variables. It is to assure norm instances to be ground, otherwise it would not be clear if the variables occurring in them should be universally or existentially quantified (see also [15]). Although we assume ground substitutions in the previous definition and some that will follow, we introduce the notion of well-formedness of a rule. Later we show that given this restriction and the semantics provided below the norm instances remain ground.

Definition 2 (Well-formedness). *We say an ic-rule $\beta(\overline{x}) \Rightarrow [ni_0(\overline{y}_0), ..., ni_n(\overline{y}_n)][ni'_0(\overline{z}_0), ..., ni'_m(\overline{z}_m)]$ is well-formed iff $\bigcup_{0 \leq j \leq n} \overline{y}_j \cup \bigcup_{0 \leq k \leq m} \overline{z}_k \subseteq \overline{x}$.*

We say an instance-revising sc-rule of the form $\beta \Rightarrow_* [ns_0, ..., ns_n][ns'_0, ..., ns'_n]$ is well-formed iff each norm scheme $ns_j = l : \langle \varphi_c, \mathbb{P}(\varphi_x(\overline{v})), \varphi_d(\overline{w}) \rangle$ and each norm scheme $ns'_j = l' : \langle \varphi'_c, \mathbb{P}'(\varphi'_x(\overline{x})), \varphi'_d(\overline{y}) \rangle$ for $0 \leq j \leq n$ we have $\overline{x} \cup \overline{y} \subseteq \overline{v} \cup \overline{w}$.

It should be noted that this restriction only guarantees the instances to remain ground in the context of the norm change mechanism. Additional restrictions are needed for the enforcement mechanism, but these are not shown here. The configuration of a normative artifact is defined as follows.

Definition 3 (Normative Artifact). *A (normative) artifact configuration is a tuple $\langle \sigma_b, \sigma_i, \Delta, \delta, R_{ic}, R_{sc} \rangle$ with:*

- *σ_b a set of ground brute facts;*

- *σ_i a set of ground insitutional facts;*
- *Δ a set of norm schemes;*
- *δ a set of ground norm instances;*
- *R_{ic} a set of well-formed ic-rules;*
- *R_{sc} a set of well-formed instance-revising sc-rules and instance-preserving sc-rules.*

A configuration $\langle \sigma_b, \emptyset, \Delta, \emptyset, R_{ic}, R_{sc}\rangle$ specified by a program s.t. σ_b is characterised by the program's facts component, Δ defined by the program's norms component, R_{ic} and R_{sc} respectively defined by the program's ic-rules and sc-rules component is called an initial artifact configuration.

Henceforth, we assume a normative artifact configuration $\langle \sigma_b, \sigma_i, \Delta, \delta, R_{ic}, R_{sc}\rangle$. R_{ic} and R_{sc} will be ommitted, because they will not change during computation.

In applying the rules of norm change their precondition needs to be evaluated. Recall that the condition ranges over brute facts, institutional facts and norm instances. We define the entailment for preconditions as follows.

Definition 4 (Entailment). *Let ϕ be a (brute or institutional) literal, $(l, \mathbb{P}(\varphi_x), \varphi_d)$ a norm instance, and $\psi_1(\overline{x})$, $\psi_2(\overline{y})$ a rule's antecedent. Given substitutions θ, θ_1 and θ_2, the entailment relation $\models_t$ that evaluates rule condition expressions w.r.t. sets of brute facts, institutional facts and norm instances $(\sigma_b, \sigma_i, \delta)$ is defined as:*

- $(\sigma_b, \sigma_i, \delta) \models_t (\phi)\theta$ iff $\phi\theta \in (\sigma_b \cup \sigma_i)$
- $(\sigma_b, \sigma_i, \delta) \models_t (l, \mathbb{P}(\varphi_x), \varphi_d)\theta$ iff $(l, \mathbb{P}(\varphi_x), \varphi_d)\theta \in \delta$
- $(\sigma_b, \sigma_i, \delta) \models_t (\psi_1(\overline{x})$ `and` $\psi_2(\overline{y}))\theta$ iff $\exists\theta_1 : [\theta_1 = \theta|\overline{x}$ and $(\sigma_b, \sigma_i, \delta) \models_t \psi_1(\overline{x})\theta_1$ and $\exists\theta_2 : [\theta_2 = \theta|(\overline{y} \setminus \overline{x})$ and $(\sigma_b, \sigma_i, \delta) \models_t \psi_2(\overline{y})\theta_1\theta_2]]$

in which "$|$" is to be read as "restricted to the domain".

An ic-rule is applicable when its precondition can be entailed for at least one substitution θ. Applying an ic-rule is then to consecutively apply each substitution that satisfies the precondition on each element of the rule's retraction and assertion list. The result is a set of norm instances to be removed and a set of norm instances to be asserted. The artifact's norm instances will then be updated by first removing the first set and then asserting the latter set. The following transition rule defines the application of an ic-rule. We annotate the transition (and the transitions that will follow) with information about the substitutions and which rule is applied. This is not essential for the semantics, but we use it later on to ease notation.

Rule 1. *Let $r = (\beta \Rightarrow [ni_0, ..., ni_n][ni'_0, ..., ni'_m]) \in R_{ic}$ in which $n, m \geq 0$ be a norm instance change rule.*

$$\frac{\Theta = \{\theta \mid (\sigma_b, \sigma_i, \delta) \models_t \beta\theta\} \quad \Theta \neq \emptyset}{\langle \sigma_b, \sigma_i, \Delta, \delta\rangle \xrightarrow{r,\Theta} \langle \sigma_b, \sigma_i, \Delta, \delta'\rangle}$$

with $\delta' = (\delta \setminus \{ni_0\theta, ..., ni_n\theta \mid \theta \in \Theta\}) \cup \{ni'_0\theta, ..., ni'_m\theta \mid \theta \in \Theta\}$

Recall that the application of an instance-preserving rule does only affect the norm schemes, leaving their associated instances unaltered. The transition rule that defines the application of instance-preserving sc-rules is defined in a similar manner as that of the ic-rules. The difference is that now the norm schemes are updated instead of the norm instances. Unlike ic-rules, we assume that each variable that occurs in an (instance-revising or instance-preserving) sc-rule's antecedent does not occur in its consequent. We take this assumption to not unnecessarily complicate the semantics of the (in particular instance-revising) sc-rules. Because of this assumption only one substitution for which the rule is applicable has to be considered.

Rule 2. *Let $r = (\beta \Rightarrow [ns_0, ..., ns_n][ns'_0, ..., ns'_m]) \in R_{sc}$ in which $n, m \geq 0$ be a norm scheme change rule.*

$$\frac{(\sigma_b, \sigma_i, \delta) \models_t \beta\theta}{\langle \sigma_b, \sigma_i, \Delta, \delta\rangle \xrightarrow{r} \langle \sigma_b, \sigma_i, \Delta', \delta\rangle}$$

with $\Delta' = (\Delta \setminus \{ns_1, ..., ns_n\}) \cup \{ns'_1, ..., ns'_m\}$

The application of an instance-revising sc-rule not only involves updating the norm schemes, but also involves updating the norm instances that have been instantiated out of them. Recall that we use the substitutions that are used for creating instances of the norm scheme to be modified, say ns, for creating new norm instances of the norm scheme ns is replaced with. To modify a norm scheme we need to know which norm instances it has created and which substitution of the variables has been used in creating them.

Definition 5 (Instances). *Let ns be a norm scheme and S be a set of norm instances. The function I that evaluates to all norm instances in S which are instances of ns together with their associatated substitutions, is defined as:*

$$\mathrm{I}(ns, S) = \{(ni, \theta) \mid ni \in S \text{ and } inst(ns, \theta) = ni \text{ and } \theta \text{ a ground substitution}\}$$

Remember that the consequent of an instance-revising sc-rule is of the form $[ns_0, ..., ns_n][ns'_0, ..., ns'_n]$ intuitively meaning that each norm scheme ns_i for $0 \leq i \leq n$ will be replaced by norm scheme ns'_i and the instances of ns_i will be reinstantiated into instances of ns'_i. Applying this instance-preserving sc-rule then boils down to: 1) removing each norm scheme ns_i from the normative artifact; 2) asserting each norm scheme ns'_i to the normative artifact; 3) removing all instances of each norm scheme ns_i; and 4) instantiating each ns'_i with the same substitutions that were used in instantiating the associated norm instances of ns_i.

Rule 3. *Let $r = (\beta \Rightarrow_* [ns_0, ..., ns_n][ns'_0, ..., ns'_n]) \in R_{sc}$ in which $n \geq 0$ be a norm scheme change rule.*

$$\frac{(\sigma_b, \sigma_i, \delta) \models_t \beta\theta}{\langle \sigma_b, \sigma_i, \Delta, \delta\rangle \xrightarrow{r} \langle \sigma_b, \sigma_i, \Delta', (\delta \setminus \delta^-) \cup \delta^+\rangle}$$

with

$$\Delta' = (\Delta \setminus \{ns_0, ..., ns_n\}) \cup (\{ns'_0, ..., ns'_n\} \setminus \{\texttt{nil}\})$$
$$\delta^+ = \bigcup_{0 \leq j \leq n} \{inst(ns'_j, \theta) \mid (ni, \theta) \in \mathrm{I}(ns_j, \delta) \text{ and } ns'_j \neq \texttt{nil}\}$$
$$\delta^- = \bigcup_{0 \leq j \leq n} \{ni \mid (ni, \theta) \in \mathrm{I}(ns_j, \delta)\}$$

We conclude this section by showing some basic, yet essential, properties the semantics exhibits. These properties summarize the meaning of the norm-change rules and demonstrate they indeed behave as we intuitively explained in previous sections. To ease notation we define the some auxiliary operators. Given an ic-rule or sc-rule rule $r = \beta \Rightarrow_{(*)} [\phi_0, ..., \phi_n][\phi'_0, ..., \phi'_m]$ and substitutions Θ we define:

$$add(r)\Theta = \{\phi'_0\theta, ..., \phi'_m\theta \mid \theta \in \Theta\}$$
$$del(r)\Theta = \{\phi_0\theta, ..., \phi_n\theta \mid \theta \in \Theta\}$$
$$tail(r) = [\phi_0, ..., \phi_n][\phi'_0, ..., \phi'_m]$$

Set of substitutions Θ will be omitted whenever empty.

Definition 6 (Update, addition and retraction). *Let $\langle \sigma_b, \sigma_i, \Delta, \delta \rangle$ be an artifact configuration. The operators $\oplus$, $\ominus$ and $\circledast$ are defined as follows:*

- $\delta \oplus add(r)\Theta = \delta'$ iff $r = \beta \Rightarrow [\,][ni_0, ..., ni_n]$ and $\langle \sigma_b, \sigma_i, \Delta, \delta \rangle \xrightarrow{r,\Theta} \langle \sigma_b, \sigma_i, \Delta, \delta' \rangle$
- $\delta \ominus del(r)\Theta = \delta'$ iff $r = \beta \Rightarrow [ni_0, ..., ni_n][\,]$ and $\langle \sigma_b, \sigma_i, \Delta, \delta \rangle \xrightarrow{r,\Theta} \langle \sigma_b, \sigma_i, \Delta, \delta' \rangle$
- $(\Delta, \delta) \circledast tail(r) = (\Delta', \delta')$ iff $r = \beta \Rightarrow_* [ns_0, ..., ns_n][ns'_0, ..., ns'_n]$ and $\langle \sigma_b, \sigma_i, \Delta, \delta \rangle \xrightarrow{r} \langle \sigma_b, \sigma_i, \Delta', \delta' \rangle$

The following propositions highlight the essential meaning of norm instance change rules. The first two show that instances of the retract list are indeed removed from the set of norm instances and that this set is indeed expanded by the instances of the addition list. The third shows that retracting and consecutively asserting a set of instances yields the same set. It tells us that we can recover a change once made. It is interesting to note the similarity (but not equivalence!) with the success of retraction and expansion, and recovery AGM postulates [1].

Proposition 1. *Given set of norm instances S.*

1. $S \subseteq (\delta \oplus S)$
2. $S \cap (\delta \ominus S) = \emptyset$
3. *if* $S \subseteq \delta$ *then* $\delta = (\delta \ominus S) \oplus S$

Sketch of Proof. *Directly follows from the definition of rule 1 which adds/removes the whole set and only the whole set of norm instances $\{ni_0\theta, ..., ni_n\theta \mid \theta \in \Theta\}$ to/from δ.*

Similar (trivial) results can be shown for the instance-preserving sc-rules, but are omitted to save space. Instead, we show similar properties for the instance-revising scheme change rules. The first proposition pertains to success of addition, whereas the second pertains to success of retraction. The third proposition could be considered success of reinstantiating the norm instances according to the change of their underlying norm scheme. The fourth proposition can be considered the recovery postulate for an instance-revising update of the norm schemes.

Proposition 2. *Given an instance-revising sc-rule $r1 = \beta \Rightarrow_* [ns_0, ..., ns_n][ns'_0, ..., ns'_n]$ and its reverse $r2 = \beta \Rightarrow_* [ns'_0, ..., ns'_n][ns_0, ..., ns_n]$ s.t. all norm schemes have disjoint labels and are not* `nil`*, and $(\delta', \Delta') = (\delta, \Delta) \circledast tail(r1)$.*

1. $add(r1) \subseteq \Delta'$
2. $del(r1) \cap \Delta' = \emptyset$
3. *if* $inst(ns_j, \theta) \in \delta$ *with* $0 \le j \le n$ *and subst.* θ *then* $inst(ns_j, \theta) \notin \delta'$ *and* $inst(ns'_j, \theta) \in \delta'$
4. *if* $del(r1) \subseteq \Delta$ *and* $add(r1) \cap \Delta = \emptyset$ *and* $\forall_{0 \le j \le n}$: $\mathrm{I}(ns'_j, \delta) = \emptyset$ *then* $(\delta, \Delta) = (\delta', \Delta') \circledast tail(r2)$

Proof. *Let $S = \{ns_0, ..., ns_n\}$ and $S' = \{ns'_0, ..., ns'_n\}$.*

(1 and 2) Follows from rule 3 in which $\Delta' = (\Delta \setminus S) \cup S'$ and S disjoint with S' (by assumption of disjoint labels).

(3) Note that $\delta' = (\delta \setminus \delta^-) \cup \delta^+$ (definition of rule 3). We have $inst(ns'_j, \theta) \in \delta'$, because $inst(ns'_j, \theta) \in \delta^+$ through 1) $inst(ns_j, \theta) \in \delta$ (by assumption) and 2) $ns'_j \neq$ `nil` *(by assumption). We have $inst(ns_j, \theta) \notin \delta'$ since 1) $inst(ns_j, \theta) \in \delta^-$ and 2) $inst(ns_j, \theta) \notin \delta^+$ (assumption of disjoint labels).*

(4) Let $(\delta'', \Delta'') = (\delta', \Delta') \circledast tail(r2)$. To prove that $(\delta, \Delta) = (\delta'', \Delta'')$. Rule 3 ensures $\Delta = \Delta''$ because 1) $\Delta' = (\Delta \setminus S) \cup S'$ (rule 3), 2) $\Delta'' = (\Delta' \setminus S') \cup S$ (rule 3), 3) $S \subseteq \Delta$ and $S' \cap \Delta = \emptyset$ (by assumption) and 4) $S \cap S' = \emptyset$ (assumption of disjoint labels). To prove $\delta = \delta''$ we show that $ni \in \delta \Leftrightarrow ni \in \delta''$ with ni an instance of ns_j or ns'_j for $0 \le j \le n$. Note that $\delta'' = (\delta' \setminus \delta'^-) \cup \delta'^+$ and $\delta' = (\delta \setminus \delta^-) \cup \delta^+$ (by definition of rule 3). Note that $\mathrm{I}(ns'_j, \delta) = \emptyset$ (by assumption), so for ($\Rightarrow$) we only take $inst(ns_j, \theta) \in \delta$ for ground substitution θ. Rule 3 ensures $inst(ns_j, \theta) \in \delta''$ because 1) $inst(ns'_j, \theta) \in \delta'$ (by definition of δ^+ and $ns'_j \neq$ `nil`*), 2) $(inst(ns'_j, \theta), \theta) \in \mathrm{I}(ns'_j, \delta')$ (because of well-formedness) and consequently 3) $inst(ns_j, \theta) \in \delta'^+$ (by definition of δ'^+ and $ns_j \neq$* `nil`*). For ($\Leftarrow$) we use contraposition. Assume $inst(ns_j, \theta) \notin \delta$ for ground substitution θ. Rule 3 ensures $inst(ns_j, \theta) \notin \delta''$, because 1) $inst(ns'_j, \theta) \notin \delta$ (by assumption), 2) $inst(ns'_j, \theta) \notin \delta'$ (by definition of δ^+ and disjoint labels) and consequently 3) $inst(ns_j, \theta) \notin \delta'^+$ (by definition of δ'^+). Next, assume $inst(ns'_j, \theta) \notin \delta$ for some substitution θ. Rule 3 ensures that $inst(ns'_j, \theta) \notin \delta''$ because 1) δ'^- is such that all instances of ns'_j are removed and 2) $inst(ns'_j, \theta) \notin \delta'^+$ (by assumption of disjoint labels).*

Interestingly, recovery only holds under certain conditions of which the least obvious one is the restriction that all labels are disjoint. (In fact, this restriction is too strong, but keeps the proposition comprehensible.) This excludes consecutively applying a rule with consequents $[ns1, ns2][ns3, ns3]$ and $[ns3, ns3][ns1, ns2]$. Suppose $ns1 = l1 : \langle c, O(x(X)), d \rangle$, $ns2 = l2 : \langle c, O(x(X)), d \rangle$ and $ns3 = l3 : \langle c, O(x(X)), d \rangle$ It is left to the reader to check that given set of norm instances $\delta = \{(l1, O(x(a)), d)\}$ this excution yields $\delta'' = \{(l1, O(x(a)), d), (l2, O(x(a)), d)\}$.

The concept of well-formedness was introduced with the goal of guaranteeing the norm instances to be ground. The following proposition shows that given the semantics for the norm change rules they indeed remain ground.

Proposition 3. *Let $\gamma_0 = \langle \sigma_b, \sigma_i, \Delta, \delta \rangle$ be an artifact configuration s.t. all norm instances in δ are ground. Then for every trace $\gamma_0 \rightarrow_* \gamma_n$ with $\gamma_n = \langle \sigma'_b, \sigma'_i, \Delta', \delta' \rangle$ it holds that that each norm instance $ni \in \delta'$ is ground.*

Proof. *Only transition rules 1 and 3 add instances. Rule 1 is such that the substitutions that satisfy the antecedent are applied on the norm instances in the consequent. These substitutions are ground because of well-formedness. Rule 3 reinstantiates the instances of each scheme in the retract list by applying the substitution used in creating them on the corresponding scheme in the assert list. Because of well-formedness these substitutions are ground.*

6. RELATED WORK

Artikis [2] has presented an infrastructure allowing agents to modify a protocol (a set of laws) at runtime. A protocol specification is stratisfied in n layers, in which layer 0 defines the domain protocol and each level $0 < k \le n$ defines a meta protocol specifying the regulations for changing the level $k - 1$ protocol. Part of a protocol specification are the "Degrees of Freedom" which define the protocol's specification components that may be modified, for example, different alternatives for a law that are replacable at runtime. Unlike Artikis we do not distinguish between different protocol levels; under which circumstances the norms in our framework may change is statically specified by the norm change rules' conditions. Another difference is related to the principle of enforcement-indepency (cf. section 2). In

Artikis' approach the changes that can be made to the protocols are hardwired in the protocols themselves, whereas in our approach the code that defines how the norms may evolve is completely separated from their specification.

In [5, 6] Bou et al. and Campos et al. have proposed an approach in which the normative framework ("electronic institution" in their terminology) can change the norms at runtime. They extend a normative framework with a set of values modeling information about environment and agents, and a set of quantative goals denoting the desired values and a transition function that specifies how the norms evolve based on the institutional goals and observed properties. The main aim of [5] is to learn the transition function that best accomplishes the institutional goals, whereas more close to our approach in [6] it is assumed that this transition function is defined by the programmer. Contrasting our approach in [5, 6] changing the norms is limited to the modification of existing norms. That is, neither new norms can be added nor can existing norms be removed. Moreover, we adopt a qualitative rather than a quantative approach in modeling information about agents and environment.

Also related to our work is the theoretical work of Boella and Van der Torre [3] who have proposed a logical framework for modeling a normative system ("normative agent" as they call it) in which "count-as rules" specify when and how the norms of the system may be changed. These count-as rules show close similarities to our norm change rules; the antecedent of the rules ranges over brute and institutional facts specifying when the norms may be changed, whereas the consequent contains actions that define how the norms should be modified.

7. CONCLUSION AND FUTURE WORK

We presented the syntax and operational semantics of generic programming constructs to facilitate the runtime modification of norms. We introduced rule-based constructs for modifying 1) conditional obligations and prohibitions (normative specification), 2) the detached obligations and prohibitions (deontic instances) they create, and 3) the normative specification such that also their associated deontic instances are automatically updated. The architecture we presented enables a programmer to specify when and how the norms may be changed by external agents or by the normative framework itself.

The operational semantics is already close to the implementation of an interpreter and allowed us to mathematically investigate some basic properties of norm change rules. Yet, to gain more insights in the practical aspects of our approach, an actual implementation is indispensable. Therefore, we are in the process of implementing a prototype in the rule-based language Jess[3]. It is also important to note that the semantics does not define any order in which the norm change rules are applied. Observe that applying norm change rules in different order might yield different results. For one reason, because the preconditions range over norm instances, the application of an ic-rule might enable or disable the application of some other rule. We envisage different strategies for applying norm change rules, e.g. an explicit priority ordering among the norm change rules. This is left for future research. Also, we did not say anything about norm consistency. As for now, the burden of avoiding conflicts between norms is on the programmer. Introducing a mechanism to guarantee norm consistency without compromising practicality is left future research. AGM-like postulates could then be used to posit the properties such a mechanism should exhibit. Other future research directions include extending the framework to deal with constitutive norms (cf. [3]) and "contrary-to-duty" norms also (cf. [15]).

[3]See `http://www.jessrules.com` for more information.

8. REFERENCES

[1] C. E. Alchourron, P. Gardenfors, and D. Makinson. On the logic of theory change: Partial meet contraction and revision functions. *Journal of Symbolic Logic*, 50:510–530, 1985.

[2] A. Artikis. Dynamic protocols for open agent systems. In *Proc. of AAMAS*, pages 97–104. IFAAMAS, 2009.

[3] G. Boella and L. van der Torre. Regulative and constitutive norms in normative multiagent systems. In *Proc. of KR*, pages 255–265. AAAI Press, 2004.

[4] G. Boella, L. van der Torre, and H. Verhagen. Introduction to normative multiagent systems. *Comput. Math. Organ. Theory*, 12(2-3):71–79, 2006.

[5] E. Bou, M. López-Sánchez, and J. A. Rodríguez-Aguilar. Adaptation of autonomic electronic institutions through norms and institutional agents. In *Proc. of ESAW*, pages 300–319, 2006.

[6] J. Campos, M. López-Sánchez, J. A. Rodríguez-Aguilar, and M. Esteva. Formalising situatedness and adaptation in electronic institutions. In *COIN IV. Revised Selected Papers*, pages 126–139, Berlin, Heidelberg, 2009. Springer-Verlag.

[7] C. Castelfranchi. Engineering social order. In *Proc. of ESAW*, pages 1–18, London, UK, 2000. Springer.

[8] M. Dastani, D. Grossi, J.-J. C. Meyer, and N. Tinnemeier. Normative multi-agent programs and their logics. In *Post. proc. of KRAMAS'08*, 2009.

[9] V. Dignum, editor. *Handbook of Research on Multi-Agent Systems: Semantics and Dynamics of Organizational Models*. IGI Global, 2009.

[10] V. Dignum. *The Role of Organization in Agent Systems*, chapter 1, pages 1–16. In Handbook of Research on Multi-Agent Systems [9], 2009.

[11] G. Governatori and A. Rotolo. Changing legal systems: Abrogation and annulment. Part I: Revision of defeasible theories. In *Proc. of DEON*, pages 3–18. Springer, 2008.

[12] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[13] A. Ricci, M. Viroli, and A. Omicini. Give agents their artifacts: the A&A approach for engineering working environments in MAS. In *AAMAS*, 2007.

[14] Y. Shoham and M. Tennenholtz. On social laws for artificial agent societies: off-line design. *Art. Int.*, 73(1-2):231–252, 1995.

[15] N. Tinnemeier, M. Dastani, J.-J. C. Meyer, and L. van der Torre. Programming normative artifacts with declarative obligations and prohibitions. In *Proc. of WI/IAT'09*. IEEE Computer Society, 2009.

[16] F. Zambonelli, N. Jennings, and M. Wooldridge. Organisational rules as an abstraction for the analysis and design of multi-agent systems. *Softw.. Eng. Knowl. Eng.*, 11(3):303–328, 2001.