Equivalence of defeasible normative systems

José Júlio Alferes, Ricardo Gonçalves, and João Leite *

CENTRIA - Dep. Informática, Faculdade de Ciências e Tecnologia,
Universidade Nova de Lisboa

Abstract. Normative systems have been advocated as an effective tool to regulate interaction in multi-agent systems. The use of deontic operators and the ability to represent defeasible information are known to be two fundamental ingredients to represent and reason about normative systems. In this paper, after proposing a framework that combines standard deontic logic and non-monotonic logic programming, deontic logic programs (DLP), we tackle the fundamental problem of equivalence between normative systems using a deontic extension of David Pearce's Equilibrium Logic and its monotonic basis, the logic of Here-and-There. We also show how deontic logic programs can be used to represent and reason about normative systems, and establish a strong connection with input-output logic.

1 Introduction

Normative systems have been advocated as an effective tool to regulate interaction in multi-agent systems. Essentially, norms encode desirable behaviours for a population of a natural or artificial society. In general, they are commonly understood as rules specifying what is expected to follow (obligations, permissions, ...) from a specific set of facts. Moreover, in order to encourage agents to act according to the norms, normative systems should also be able to specify the application of rewards/sanctions.

A deontic logic is a logic that deals with the notions of obligation and permission, and it is, therefore, a fundamental tool for modeling normative reasoning. Since the seminal work of von Wright [33] many have investigated and developed systems of deontic logic [13]. One such system is the modal logic KD, usually known as Standard Deontic Logic (SDL) [6]. Although accepted as a fundamental tool for modeling normative assertions, SDL has shown not to be enough for the task of representing norms [7]. For instance, its inability to deal with some paradoxes is well known, namely paradoxes involving the so-called contrary-to-duty obligations. The main difficulty of SDL is the fact that classical implication does not provide a faithful representation for the conditional obligations that usually populate a normative system.

* The sencond author was supported by FCT under the postdoctoral grant SFRH/BPD/47245/2008. The work was partially supported by projects ERRO – PTDC/EIA-CCO/121823/2010, and ASPEN – PTDC/EIA-CCO/110921/2009.

Several approaches modeling conditional obligations have been proposed and shown to have a more reasonable behavior in the face of the aforementioned paradoxes. These approaches include, for example, works using dyadic modal logics [32, 16] and input-output logic [20].

Another fundamental ingredient for modeling norms is the ability to express defeasible knowledge. This is important for representing exceptions, which are very common in normative rules. Several approaches using non-monotonic logics where applied to the problem of representing and reasoning about norms. Important works on this topic include, for example, the seminal work of Sergot et al. [29], Prakken and Sartor [26], Governatori et al. [3] and Brewka [4].

For all the reasons aforementioned, the representation and reasoning about normative systems would greatly benefit from a framework combining deontic logic and rule based non-monotonic reasoning. It is not surprising, therefore, that some attention has been devoted to the combination of these two notions [23]. Important approaches include the works of McCarty [22] and Lee and Ryu [28], the work of Governatori [12] using deontic rules in defeasible logic, the work of Horty [14], Makinson and van der Torre work using input-output logic [19], and the recent work of Alberti et al using hybrid knowledge bases [2, 1]. Nevertheless, none of the above approaches succeeds in complying with all of the following syntactical and semantical requirements: from a syntactical point of view, a framework for specifying and reasoning with normative systems should have a rich language, thus allowing complex deontic formulas to appear both in the body and in the head of rules. With respect to the semantics, such normative framework should have a declarative semantics. This would allow the agents (the ones that are subject to the normative system), the modeler (the one that writes down the norms) and the electronic institution (the one responsible for monitoring the agents and applying the sanctions/rewards) to reason about the normative system in a simple and clear way. Moreover, fundamental notions such as equivalence, redundancy and consistency should be easily defined in the framework.

In this paper we start by proposing a language for representing and reasoning about normative systems that combines deontic logic with logic programming with default negation. We give it a fully declarative semantics, allowing the definition of a strong notion of equivalence between normative systems.

Two major features distinguish our approach from other formalisms that combine deontic operators with non-monotonic reasoning. First of all, we have a rich language, which allows complex deontic logic formulas to appear in the body and in the head of a rule, combined with the use of default negation. Moreover, at the level of the semantics, we endow the normative systems with a purely declarative semantics, which stems from the stable model semantics of logic programs. Furthermore, by making use of David Pearce's results on Equilibrium Logic [24] and its relation to strong equivalence in logic programs [17], we can define the fundamental notion of equivalence between normative systems, and, more importantly, we develop a logic which allows us to verify equivalence of normative systems by using logical equivalence.

We end the paper by showing that we can embed an important fragment of input-output logic [19] in our framework.

The paper is structured as follows: in Section 2 we introduce the framework for normative systems, along with some examples. Then, in Section 3, we define the semantics of normative systems and revisit the examples. The study of the notion of equivalence of normative systems is done in Section 4. In Section 5 we prove an embedding result for input-output logic. Finally, in Section 6, we draw some conclusions and point out paths for future research.

Acknowledgements

Having worked, over the years, on Logic Programming for Non-Monotonic Reasoning, and Answer-Set Programming in particular, the work of David on Equilibrium Logic and its relationship with ASP was, for us as for everyone else in this area, of great importance.

But, to us, David has been more than *the guy who discovered the relationship between the Logic of Here-and-There and ASP*. He has been a great colleague, always active in bringing people together, and a very good friend with whom we have shared many great moments. We are looking forward to his next 60 years.

2 Framework

In this section we introduce the main ingredients for setting up a normative framework that jointly combines the expressivity of deontic logic with that of non-monotonic rules from logic programming. We start by briefly introducing standard deontic logic.

2.1 Standard deontic logic

The formal study of deontic logic was highly influenced by modal logic. In fact, standard deontic logic (SDL) is a modal logic with two modal operators, one for obligation and another for permission. We briefly describe SDL, and we refer to [6] for further details.

Formally, the language of SDL, dubbed L_{SDL}, is constructed from a set $Prop$ of propositional symbols using the usual classical connectives $\sim, \Rightarrow$, and the unary deontic operator $\mathbf{O}$ (obligation). The classical connectives $\vee$ and $\wedge$ can be defined as abbreviations in the usual way. Moreover, the permission operator can be defined as $\mathbf{P} := \sim \mathbf{O} \sim$.

The semantics of SDL is a Kripke-style semantics. A Kripke model is a tuple $\langle W, R, \mathcal{V} \rangle$, where W is a set, the possible worlds, $R \subseteq W \times W$ is the accessibility relation, and $\mathcal{V} : W \to 2^{Prop}$ is a function assigning, to each world, the set of propositional symbols true in that world. Moreover, the relation R is assumed to be serial, i.e., for every $w \in W$ there exists $w' \in W$ such that wRw'. As usual, we define the satisfaction of a formula φ in a Kripke model $\mathcal{M} = \langle W, R, \mathcal{V} \rangle$ at a world $w \in W$, by induction on the structure of φ.

i) $\mathcal{M}, w \Vdash p$ if $p \in \mathcal{V}(w)$, for $p \in Prop$;
ii) $\mathcal{M}, w \Vdash\sim \varphi$ if $\mathcal{M}, w \nVdash \varphi$;
iii) $\mathcal{M}, w \Vdash \varphi_1 \Rightarrow \varphi_2$ if $\mathcal{M}, w \nVdash \varphi_1$ or $\mathcal{M}, w \Vdash \varphi_2$;
iv) $\mathcal{M}, w \Vdash \mathbf{O}(\varphi)$ if $\mathcal{M}, w' \Vdash \varphi$ for every w' such that $\langle w, w' \rangle \in R$.

We say that an SDL formula φ is a *logical consequence* of a set Φ of SDL formulas, written $\Phi \vdash_{SDL} \varphi$, if for every Kripke model $\mathcal{M} = \langle W, R, \mathcal{V} \rangle$ and every world $w \in W$ we have that $\mathcal{M}, w \Vdash \varphi$ whenever $\mathcal{M}, w \Vdash \delta$ for every $\delta \in \Phi$.

A formula φ is said to be a SDL *theorem* if $\emptyset \vdash_{SDL} \varphi$.

At this point it is important to stress that we are using the so-called local consequence relation, contrasted with the global consequence relation defined as $\Phi \vdash_g \varphi$ if for every Kripke model $\mathcal{M} = \langle W, R, \mathcal{V} \rangle$ we have that $\mathcal{M}, w \Vdash \varphi$ for every world $w \in W$ whenever $\mathcal{M}, w \Vdash \delta$ for every world $w \in W$ and every $\delta \in \Phi$. These two consequence relations are quite different, and the reason why sometimes this difference is neglected is because the interest is only on the set of theorems, which is the same for both consequences. In our approach, since the consequence relation is a fundamental tool, we do not neglect this difference and work with the local consequence which is more adequate for normative reasoning. From our point of view, the global consequence does not faithfully represent normative reasoning. This can be witnessed, for example, by the fact that we have $\varphi \vdash_g \mathbf{O}(\varphi)$ which is not a valid reasoning if $\mathbf{O}$ is to be read as an obligation.

Let us now continue by defining the notion of logical theory. As we will see, logical theories will play a fundamental role in the definition of the semantics for normative systems.

Definition 1. *A set of SDL formulas Φ is said to be a SDL logical theory if Φ is closed under SDL consequence, i.e., for every $\varphi \in L_{SDL}$ if $\Phi \vdash_{SDL} \varphi$ then $\varphi \in \Phi$.*

We denote by $Th(SDL)$ the set of theories of SDL. A fundamental property of the set of theories is that the tuple $\langle Th(SDL), \subseteq \rangle$ is a complete lattice with smallest element the set $Theo(SDL)$ of theorems of SDL and greatest element the set L_{SDL} of all SDL formulas. Given a subset A of L_{SDL} we denote by $A^{\vdash_{SDL}}$ the smallest SDL theory that contains A.

2.2 Deontic logic programs

Deontic logic programs are composed by rules that resemble usual logic program rules[1], but where complex SDL formulas can appear in the place where only atoms where allowed.

Definition 2. *A deontic logic program is a set of rules*

$$\varphi \leftarrow \psi_1, \ldots, \psi_n, not\ \delta_1, \ldots, not\ \delta_m$$

where $\varphi, \psi_1, \ldots, \psi_n, \delta_1, \ldots, \delta_m \in L_{SDL}$.

[1] For an overview of logic program rules see, e.g., [18].

As usual, the symbol $\leftarrow$ represents rule implication, the symbol "," represents conjunction and the symbol *not* represents default negation. A rule $\varphi \leftarrow \psi_1, \ldots, \psi_n, not\ \delta_1, \ldots, not\ \delta_m$ has the usual reading that φ should hold whenever $\psi_1, \ldots, \psi_n$ hold and $\delta_1, \ldots, \delta_m$ are not known to hold. We should note that these deontic logic programs can be seen as a particular case of the general construction presented in [10].

A *definite deontic logic program* is a set of rules without negations as failure, i.e. of the form $\varphi \leftarrow \psi_1, \ldots, \psi_n$.

Note that, contrarily to some works in the literature [12, 19, 14], deontic formulas can appear both in the head and in the body of a rule and, moreover, they can be complex formulas and not just atomic formulas. This extra flexibility is relevant, for example, to deal with non-compliance and application of sanctions. We can use the rule

$$\mathbf{O}(payFine \vee (apologize \wedge paySame)) \leftarrow \mathbf{O}(pay), not\ pay$$

to express that if an agent has the obligation to pay some bill, and it is not known that the agent has payed it, then the agent is obliged to pay a fine or apologize and pay the same amount.

2.3 Normative systems

We now define the central concept of normative system. A normative system is usually understood as a set of rules that specify what obligations and permissions follow from a given set of facts, and, moreover, specify sanction/rewards. In our approach, we directly use the deontic logic programs introduced in the previous section to represent a normative system.

Definition 3. *A normative system $\mathcal{N}$ is a deontic logic program.*

Another important notion is that of a set of facts[2]. A set of facts not only represents the state of the environment that is populated by the agents (using formulas without deontic operators), but also the normative state (using formulas with deontic operators).

Definition 4. *A set of facts is a set $\mathcal{F}$ of SDL formulas.*

In what follows, we abuse notation and will often write $\mathcal{F}$ to refer to its correspondent set of deontic logic program's rules $\{\varphi \leftarrow : \varphi \in \mathcal{F}\}$.

In normative multi-agent systems [30] there is usually the distinction between the so-called *brute facts*, which represent the state of the environment shared by the agents, and the *institutional facts*, which represent the normative/institutional state of the multi-agent system. Although in this paper we have adopted a simplified notion of set of facts - by not distinguishing between brute and institutional facts - the richness of our language allows us to incorporate this distinction easily.

[2] In this work we use the word *fact* with its usual meaning in the literature of normative multi-agent systems. This should not be confused with the logic programming notion of *fact*.

2.4 Examples

Now that we have defined an expressive language for representing norms, we illustrate its use in some examples. Of course, the examples, and especially the way they are handled by our framework, can only be fully understood after the definition of the semantics. However, we opt for presenting them already here, even if the expected results are only informally spelled out, so that the reader can understand what the language can be used for.

Example 1. Consider the following normative statement addapted from [27].

You should have neither a fence nor a dog. But, if you have a dog you should have both a fence and a warning sign. In a situation where you have a dog what obligations should hold?

This statement is an example of a contrary-to-duty paradox. Contrary-to-duty paradoxes are very important in the area of deontic reasoning. Not only were they crucial for revealing some of the weaknesses of SDL in modeling norms, but, more importantly, they provided fundamental intuitions for the extensions of SDL that overcame some of these weaknesses. In a nutshell, contrary-to-duty paradoxes encode the problem of what obligations should follow from a normative system in a situation where some of the existing obligations are already being violated.

The following is a first attempt to represent the above normative statement using our framework.

$$\mathbf{O}(\sim dog \wedge \sim fence) \leftarrow$$
$$\mathbf{O}(fence \wedge warningSign) \leftarrow dog$$

The problem is that, intuitively, there are circumstances in which this normative system can lead to inconsistency. In fact, if $\mathcal{F} = \{dog\}$, the conflicting obligations $\mathbf{O}(\sim dog \wedge \sim fence)$ and $\mathbf{O}(fence \wedge warningSign)$ both follow from the normative system. This reading is in accordance with, for example Prakken and Sergot [27].

If we take a closer look at the description of the problem we can see that the first rule of the normative system wrongly does not distinguish between the two obligations appearing there. While the obligation not to have a dog is unconditional, the obligation not to have a fence is not. It has an exception: the case where you have a dog. Therefore, a proper representation should use default negation to model this exception.

$$\mathbf{O}(\sim dog) \leftarrow$$
$$\mathbf{O}(\sim fence) \leftarrow not\ dog$$
$$\mathbf{O}(fence \wedge warningSign) \leftarrow dog$$

Intuitively, the above normative system never yields an inconsistency, since the rules for $\mathbf{O}(\sim fence)$ and $\mathbf{O}(fence \wedge warningSign)$ now have bodies that cannot hold at the same time (dog and $not\ dog$). Moreover, given the set of facts $\mathcal{F} = \{dog, fence\}$ the consequences of $\mathcal{N} \cup \mathcal{F}$ include $\{dog, fence, \mathbf{O}(\sim$

$dog), \mathbf{O}(fence)$, $\mathbf{O}(warningSign)\}$. Therefore, on the one hand, we are able to detect a violation of the obligation not to have a dog, and, on the other hand, the fact that we have a fence is not a violation, because the fact that there is a dog prevents the derivation of the obligation not to have a fence. We argue that this kind of reasoning is relevant. Consider, for example, that there are rules for applying sanctions in case of violations, i.e., we augment $\mathcal{N}$ with the rules $\mathbf{O}(fineD) \leftarrow \mathbf{O}(\sim dog), dog$ and $\mathbf{O}(fineF) \leftarrow \mathbf{O}(\sim fence), fence$. Then, given the set of facts $\mathcal{F} = \{dog, fence\}$, the obligation $\mathbf{O}(fineD)$ is entailed by the normative system but $\mathbf{O}(fineF)$ is not.

Example 2. Consider now an example of a online store that sells musical DVDs. It has registered clients that can order DVDs. The premium clients are granted a discount, and regular clients can become premium clients by buying a card. Since the store has expenses whenever a client makes an order, it needs a normative mechanism to enforce some fine to those clients that have ordered some product but didn't pay it. Consider the following normative system $\mathcal{N}$ representing the store's normative rules.

$$
\begin{aligned}
premium(X) &\leftarrow buyCard(X), client(X)\\
discount(X,Y) &\leftarrow client(X), product(Y), premium(X)\\
\mathbf{O}(pay(X,Y)) &\leftarrow client(X), product(Y), ordered(X,Y)\\
\mathbf{O}(pay200(X,Y)) &\leftarrow \mathbf{O}(pay(X,Y)), not\ discount(X,Y)\\
\mathbf{O}(pay100(X,Y)) &\leftarrow \mathbf{O}(pay(X,Y)), discount(X,Y)\\
paid(X,Y) &\leftarrow \mathbf{O}(pay200(X,Y)), pay200(X,Y)\\
paid(X,Y) &\leftarrow \mathbf{O}(pay100(X,Y)), pay100(X,Y)\\
\mathbf{O}(fine(X,Y)) &\leftarrow \mathbf{O}(pay(X,Y)), not\ paid(X,Y).
\end{aligned}
$$

The outcome of the normative system depends, of course, on the actual set of facts. Consider, for example, $\mathcal{F} = \{client(Bob), product(U2), ordered(Bob, U2)\}$, i.e., Bob ordered a U2 DVD but he didn't pay anything. Then $\mathbf{O}(fine(Bob, U2))$ should follow from the normative system Now suppose that Bob ordered a U2 DVD and payed only 100€, i.e, $\mathcal{F} = \{client(Bob), product(U2), ordered(Bob, U2), pay100(Bob)\}$. In this case $\mathbf{O}(fine(Bob, DVD))$ should also follow from the normative system.

Suppose that Bob ordered a U2 DVD and payed 200€, i.e, $\mathcal{F} = \{client(Bob), product(U2), ordered(Bob, U2), pay200(Bob)\}$. In that case $\mathbf{O}(fine(Bob, DVD))$ should no longer follow from the normative system. In the same way, the obligation $\mathbf{O}(fine(Bob, U2))$ should not follow in case $\mathcal{F} = \{client(Bob), product(U2), ordered(Bob, U2), pay100(Bob), buyCard(Bob)\}$, i.e, Bob ordered the U2 DVD, bought the premium client card and payed 100€, taking advantage of his premium client discount.

3 Semantics

In order to allow agents and institutions to reason about a normative system, it is very important that it has a rigorous formal semantics which, at the same time, should be clean and as simple as possible.

In this section we present a declarative semantics for normative systems, by defining a stable models like semantics [9] for deontic logic programs.

3.1 Stable model semantics

The definition of a semantics for deontic logic programs is not straightforward due to their complex language. Recall that a deontic logic program, instead of atoms, can have complex SDL formulas in the head and body of its rules. The problem is that, contrarily to the case of atoms, these formulas are not independent. A key idea to overcome this difficulty is to define a notion of interpretation that accounts for such interdependence between these "complex atoms".

Definition 5. *An interpretation is a theory T of SDL.*

Recall that a theory of SDL is a set of SDL formulas that is closed under SDL logical consequence. The key idea of taking theories of SDL as interpretations, contrasts with the usual definition of an interpretation as any set of atoms, allows the semantics to cope with the interdependence between the SDL formulas appearing in the rules.

Definition 6. *An interpretation T satisfies a rule*

$$\varphi \leftarrow \psi_1, \ldots, \psi_n, not\ \delta_1, \ldots, not\ \delta_m$$

if $\varphi \in T$ whenever $\psi_i \in T$ for every $i \in \{1, \ldots, n\}$ and $\delta_j \notin T$ for every $j \in \{1, \ldots, m\}$.

An interpretation is a *model* of a deontic logic program $\mathcal{P}$ if it satisfies every rule of $\mathcal{P}$. We denote by $Mod(\mathcal{P})$ the set of models of $\mathcal{P}$. The ordering over interpretations is just set theoretical inclusion and the notions of minimal and least interpretations are defined as usual.

To assign semantics to deontic logic programs we start, as it is usually done, by assigning semantics to definite deontic logic programs, i.e, programs without default negation. Recall that the stable model of a definite logic program is its least model. To define a similar semantics for definite deontic logic program, we need first to prove the following theorem:

Theorem 1. *Every definite deontic logic program $\mathcal{P}$ has a least model, which we denote by least($\mathcal{P}$).*

Proof. Consider the set $S_{\mathcal{P}} = \bigcap_{T \in Mod(\mathcal{P})} T$. Note that $S_{\mathcal{P}}$ always exists since $\langle Th_{SDL}, \subseteq \rangle$ is a complete lattice. Moreover, it is clear that $S_{\mathcal{P}}$ is included in every model of $\mathcal{P}$ and it is trivial to prove that $S_{\mathcal{P}}$ is still a model of $\mathcal{P}$.

To define the stable model semantics of a deontic logic program we use a Gelfond-Lifschitz like operator.

Definition 7. *Let $\mathcal{P}$ be a deontic logic program and T an interpretation. The GL-transformation of $\mathcal{P}$ modulo T is the program $\frac{\mathcal{P}}{T}$ obtained from $\mathcal{P}$ by:*

- *removing from $\mathcal{P}$ all rules which contain a literal not φ such that $\varphi \in T$;*
- *removing from the remaining rules all default negated literals.*

Since $\frac{\mathcal{P}}{T}$ is a definite deontic program, it has an unique least model. We define $\Gamma_{\mathcal{P}}(T) = least\left(\frac{\mathcal{P}}{T}\right)$.

Definition 8. *An interpretation T is a stable model of a deontic logic program $\mathcal{P}$ if $\Gamma_{\mathcal{P}}(T) = T$. We denote by $SM(\mathcal{P})$ the set of all stable models of $\mathcal{P}$. A SDL formula φ is true under the stable model semantics of $\mathcal{P}$, denoted by $\mathcal{P} \models_{SM} \varphi$, if it belongs to every stable model of $\mathcal{P}$.*

We can use the semantics of deontic logic programs to define the semantics of normative systems given a particular set of facts.

Definition 9. *Let $\mathcal{N}$ be normative system and $\mathcal{F}$ a set of facts. Then, a SDL formula φ is said to be a consequence of $\mathcal{N}$ given $\mathcal{F}$ if $\mathcal{N} \cup \mathcal{F} \models_{SM} \varphi$. We denote by $SM_{\mathcal{F}}(\mathcal{N})$ the set of all such consequences.*

Now that we have defined a semantics for normative systems, we discuss three important related concepts: equivalence, redundancy and consistency.

Equivalence The notion of equivalence is very important in the area of logic programming. Usually two logic programs are dubbed equivalent if they have the same stable model semantics. Nevertheless, it became evident that this definition of equivalence is too weak in the sense that it is not immune to the addition of contexts. For example, consider given a program $\mathcal{P}$ and suppose we want to replace part of $\mathcal{P}$ with an equivalent set of rules. There is no guarantee that the resulting program is equivalent to $\mathcal{P}$. Therefore, a stronger notion of equivalence between logic programs, dubbed strong equivalence, was defined precisely as equivalence under the presence of any context [17].

We now generalize this notion of strong equivalence to deontic logic programs.

Definition 10. *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two deontic logic programs. We say that $\mathcal{P}_1$ and $\mathcal{P}_2$ are strongly equivalent, denoted by $\mathcal{P}_1 \equiv_s \mathcal{P}_2$, if for any deontic logic program $\mathcal{P}$, the programs $\mathcal{P}_1 \cup \mathcal{P}$ and $\mathcal{P}_2 \cup \mathcal{P}$ have the same stable models.*

This notion of strong equivalence provides a robust definition of equivalence between normative systems. Robust in the sense that two equivalent normative systems have the same semantics even in the presence of an additional set of rules. This robustness can also be seen in a different, but equally important, perspective. Imagine that we have a normative system and we want to substitute part of it by an equivalent simplification. In that case, only the use of strong

equivalence guarantees that the semantics of the entire normative system is not affected by this change.

As it is usual in other notions of equivalence of normative systems (for example [19]), it is not feasible to check for strong equivalence due to the universal quantification it involves. It would be nice to have a mechanism to prove equivalence of two normative systems by just comparing them. In fact, contrary to other approaches, in Section 4.1, inspired by David Pearce's results on strong equivalence of logic programs, we define a logic in which we can check strong equivalence of deontic logic programs just by checking logical equivalence.

In some situations the notion of strong equivalence may be seen as too strong, e.g. when there are restriction on the kind of predicates describing the environment (i.e. those that can be added as facts). Interestingly, in logic programming there is a weaker notion of equivalence, called *uniform equivalence* [8, 25], which restricts the contexts that can be added to the programs. We generalize this notion of uniform equivalence to the framework of deontic logic programs.

Definition 11. *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two deontic logic programs. We say that $\mathcal{P}_1$ and $\mathcal{P}_2$ are uniformly equivalent, denoted by $\mathcal{P}_1 \equiv_u \mathcal{P}_2$, if for any set $\mathcal{F}$ of deontic formulas, the programs $\mathcal{P}_1 \cup \{\varphi \leftarrow : \varphi \in \mathcal{F}\}$ and $\mathcal{P}_2 \cup \{\varphi \leftarrow : \varphi \in \mathcal{F}\}$ have the same stable models.*

Note that, clearly, two strongly equivalent deontic logic programs are also uniformly equivalent.

Redundancy Using the notion of strong equivalence we can define the important notion of redundancy.

Definition 12. *Let $\mathcal{P}$ be a deontic logic program and let r be a rule of $\mathcal{P}$. Then, r is redundant with respect to program $\mathcal{P}$ if $\mathcal{P} \equiv_s \mathcal{P} \setminus \{r\}$.*

Redundancy is important for normative systems simplification. If a rule is redundant with respect to a normative system, we can safely remove it without affecting the semantics of the normative system.

Consistency Another important property of a normative system is consistency. When can we say that a normative system is consistent with a set of facts? Here we adopt a definition which in line with the one proposed in [19].

Definition 13. *Given a normative system $\mathcal{N}$ and a consistent set of facts $\mathcal{F}$, we say that $\mathcal{N}$ is consistent with $\mathcal{F}$ if the deontic logic program $\mathcal{N} \cup \mathcal{F}$ has a consistent stable model. A normative system $\mathcal{N}$ is* consistent *if for any consistent set of facts $\mathcal{F}$, $\mathcal{N}$ is consistent with $\mathcal{F}$.*

Let us clarify this definition with the help of an example. Consider the first normative system presented in Example 1,

$$\mathcal{N} = \{\mathbf{O}(\sim dog \wedge \sim fence) \ \leftarrow, \mathbf{O}(fence \wedge warningSign) \leftarrow dog\}.$$

Clearly, there is a conflict between the two heads of the rules whenever dog is the case. In fact, if we take the consistent set $\mathcal{F} = \{dog\}$, the program $\mathcal{N} \cup \mathcal{F}$ has only one stable model, the set of all SDL-formulas, which is not a consistent set. Therefore, according to our definition, this normative system is not consistent with $\mathcal{F} = \{dog\}$.

Consider now the second normative system of Example 1,

$$\mathcal{N} = \{\mathbf{O}(\sim dog) \leftarrow, \mathbf{O}(\sim fence) \leftarrow not\ dog, \mathbf{O}(fence \wedge warningSign) \leftarrow dog\}.$$

In this case, although there is a conflict between the head of the rules, it is clear that the bodies of these rules cannot be true at the same time. Therefore, $\mathcal{N}$ is a normative system which is consistent not only with the set $\mathcal{F} = \{dog\}$, but also with any set of facts containing only formulas without deontic operators.

3.2 Examples revisited

We now revisit the examples of Section 2.4 and present the semantics of the normative system introduced there.

Example 3. Consider the normative system of Example 1. Suppose that $\mathcal{F} = \{dog\}$ is the current set of facts. Then $SM_{\mathcal{F}}(\mathcal{N}) = \{dog, \mathbf{O}(\sim dog), \mathbf{O}(fence), \mathbf{O}(warningSign)\}^{\vdash_{SDL}}$. Suppose now that it is not known whether you have a dog, i.e, $\mathcal{F} = \{\}$. Then $SM_{\mathcal{F}}(\mathcal{N}) = \{\mathbf{O}(\sim dog), \mathbf{O}(\sim fence)\}^{\vdash_{SDL}}$, i.e., the obligation $\mathbf{O}(\sim fence)$ now follows from the normative system and $\mathbf{O}(warningSign)$ no longer follows.

Example 4. Consider the normative system of Example 2. Suppose that $\mathcal{F} = \{client(Bob), product(U2), ordered(Bob, U2)\}$, i.e., Bob ordered a U2 DVD but he didn't pay anything. Then, $SM_{\mathcal{F}}(\mathcal{N}) = \{\mathbf{O}(pay(Bob, U2)), \mathbf{O}(pay200(Bob, U2)), \mathbf{O}(fine(Bob, U2)\}^{\vdash_{SDL}}$. As expected, $\mathbf{O}(fine(Bob, U2))$ is a consequence of the normative system.

Suppose now that $\mathcal{F} = \{client(Bob), product(U2), ordered(Bob, U2), pay100(Bob), buyCard(Bob)\}$, i.e, Bob ordered the U2 DVD, bought the premium client card and payed 100€. Then, $SM_{\mathcal{F}}(\mathcal{N}) = \{client(Bob), product(U2), ordered(Bob, U2), pay100(Bob), buyCard(Bob), premium(Bob), discount(Bob, U2), \mathbf{O}(pay(Bob, U2), \mathbf{O}(pay100(Bob, U2), paid(Bob, U2)\}^{\vdash_{SDL}}$.

4 Equivalence of normative systems

Due the explicit universal quantification, it is not feasible to prove the strong equivalence of two normative systems by direct use of the definition. As we will see, the same phenomena appears, for example, in the context of input-output logics [19]. In this section we define a deontic extension of the logic of here-and-there (HT) [17], and prove that logical equivalence in this logic provides a necessary and sufficient condition for checking strongly equivalence of normative systems. This construction can be seen as a particular case of the general construction presented in [11].

4.1 Deontic equilibrium logic

Equilibrium logic [24], and its monotonic base – the logic of here-and-there (HT) – provide a logical foundation for the stable models semantics of logic programs [9], in which several important properties of logic programs under the stable model semantics can be studied. In particular, it can be used to check strong equivalence of programs by checking logical equivalence in HT (cf. [17]).

In this section we define a deontic version of HT, dubbed HT_{SDL}. The key idea is that a deontic logic program can be naturally translated to a set of formulas in HT_{SLD}, one for each rule of the program, and strong equivalence of two deontic logic programs can be checked by proving logical equivalence of the corresponding sets of formulas.

The language of HT_{SDL} is build constructively from the formulas of SDL using the connectives $\bot, \sqcap, \sqcup$ and $\sqsupset$. Negation $\neg$ is introduced as an abbreviation $\neg\delta := (\delta \to \bot)$. Note that the formulas of SDL act as atoms of the HT_{SDL} language. A rule $\varphi \leftarrow \psi_1, \ldots, \psi_n, not\ \delta_1, \ldots, not\ \delta_m$ of a deontic program is identified with the HT_{SDL}-formula $(\psi_1 \sqcap \ldots \sqcap \psi_n \sqcap \neg\delta_1 \sqcap \ldots \sqcap \neg\delta_m) \sqsupset \varphi$.

The semantics of HT_{SDL} is a generalization of the intuitionistic Kripke semantics of HT. A frame for HT_{SDL} is a tuple $\langle W, \leq\rangle$ where W is a set of exactly two worlds, say h (here) and t (there) with $h \leq t$. An HT_{SDL} interpretation is a frame together with an assignment i that associates to each world a theory of SDL, such that $i(h) \subseteq i(t)$. It is convenient to see a HT_{SDL} interpretation as an ordered pair $\langle T^h, T^t\rangle$ such that $T^h = i(h)$ and $T^t = i(t)$ where i is the interpretation's assignment. We define the satisfaction relation between an HT_{SDL} interpretation $I = \langle T^h, T^t\rangle$ at a particular world w and a HT_{SDL} formula δ recursively:

i) for $\varphi \in L_{SDL}$ we have that $\langle T^h, T^t\rangle, w \Vdash \varphi$ if $T^w \vdash_{\mathcal{L}} \varphi$;
ii) $I, w \nVdash \bot$;
iii) $I, w \Vdash (\delta_1 \sqcup \delta_2)$ if $\langle T^h, T^t\rangle, w \Vdash \delta_1$ or $\langle T^h, T^t\rangle, w \Vdash \delta_2$;
iv) $I, w \Vdash (\delta_1 \sqcap \delta_2)$ if $\langle T^h, T^t\rangle, w \Vdash \delta_1$ and $\langle T^h, T^t\rangle, w \Vdash \delta_2$;
v) $I, w \Vdash (\varphi \sqsupset \psi)$ if $\forall_{w \leq w'}$ we have $I, w' \nVdash \varphi$ or $I, w' \Vdash \psi$.

We say that an interpretation $\mathcal{I}$ is a model of an HT_{SDL} formula δ if $\mathcal{I}, w \Vdash \delta$ for every $w \in \{h, t\}$. A formula δ is said to be a consequence of a set of formulas Φ, denoted by $\Phi \vdash_{HT_{SDL}} \delta$, if for every interpretation $\mathcal{I}$ and every world w we have that $\mathcal{I}, w \Vdash \delta$ whenever $\mathcal{I}, w \Vdash \delta'$ for every $\delta' \in \Phi$. Two sets of formulas Φ_1 and Φ_2 are said to be equivalent if $\Phi_1 \vdash_{HT_{SDL}} \psi$ for every $\psi \in \Phi_2$ and $\Phi_2 \vdash_{HT_{SDL}} \psi$ for every $\psi \in \Phi_1$.

The following is the main theorem of this section. Its proof can be obtained by adapting the general result proved in [11]. It allows us to prove strong equivalence of two deontic logic programs by checking if the two programs are logically equivalent in HT_{SDL}.

Theorem 2. *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two deontic logic programs. Then, $\mathcal{P}_1$ and $\mathcal{P}_2$ are strongly equivalent iff $\mathcal{P}_1$ and $\mathcal{P}_2$ are logically equivalent in HT_{SDL}.*

5 Embedding Input-output Logic

In this section we show that the deontic logic programs are expressive enough to embed input-output logic. As a consequence, in the end of the section we show that deontic HT is a sound tool to check equivalence in the sense of input-output logic. We start by introducing input-output logic.

5.1 Input-Output Logic

The key idea in input-output logic (IO logic) [19] is to represent norms using pairs of formulas, rather than with just formulas, as it is usual in deontic logics. The central elements in the language of IO logic are, therefore, the pairs $\langle \varphi, \psi \rangle$, where φ and ψ are classical propositional formulas. Intuitively, a pair $\langle \varphi, \psi \rangle$ represents the conditional norm that whenever the body φ (the input) is true then the head ψ (the output) is obligatory. As an example, the pair

$$\langle driving \wedge redSignal, stop \rangle$$

can be seen as the representation of the norm stating that, whenever you are driving and there is a red signal, you have the obligation to stop.

Definition 14. *A generating set is a set G of pairs of propositional formulas.*

Generating sets can be seen as the formal representation of a normative code, i.e., a set of conditional norms. The term generating set comes from the intuition that it generates the output from a given input. An *input* is a set A of propositional formulas.

Given a generating set G and an input A, we consider the set

$$G(A) = \{\psi : \langle \varphi, \psi \rangle \in G \text{ and } \varphi \in A\}.$$

Intuitively, the set $G(A)$ can be seen as the direct consequences of the normative system G given that the formulas of the input A hold. This construction of $G(A)$, however, does not take into account the logical interdependence between formulas. For example, if $G = \{\langle p, q \rangle\}$ and $A = \{p \wedge r\}$ then $q \notin G(A)$. To cope with this interdependency the so-called *out* operations were defined. The idea is that the *out* operations represent the different ways in which the logical interdependence between formulas can be handled.

The semantics of IO logic is an operational semantics which is parametrized by the choice of this *out* operations. Operation $out(G, A)$ takes a generating set G and an input A and returns a (output) set of formulas. Four natural *out* operations are usually considered: the simple-minded operator out_1, the basic operator out_2, reusable operator out_3, and reusable basic operator out_4.

Here, we focus on two of these operators[3]: out_1 and out_3. Given a set A of formulas we denote by $Cn(A)$ the set of consequence of A in classical logic. Recall that a classical theory is a set T such that $Cn(T) = T$. We can now define the *out* operations.

[3] We do not consider all four operators because the other two are less interesting, and considering them would turn the definitions lengthy, almost repetitive, and not particularly more interesting.

Definition 15. *Given a generating set G and an input A, we define:*

- $out_1(G, A) = Cn(G(Cn(A)))$
- $out_3(G, A) = \bigcap\{Cn(G(B)) : A \subseteq B,\ B = Cn(B), and\ G(B) \subseteq B\}$

Moreover, for each operator out_n, we can consider out_n^+, the correspondent throughput operator that allows inputs to reappear as outputs. These operators are defined as

$$out_n^+(G, A) = out_n(G \cup Id, A)$$

where the Id is the identity binary relation, i.e., is defined as

$$Id = \{\langle\varphi, \varphi\rangle : \varphi \text{ classical formula}\}.$$

In what follows, we use $out(G, A)$ to refer to any of the above *out* operations.

Although the above formulation of IO logic already gives a reasonable account of conditional obligations, it is not enough for reasoning with contrary-to-duty situations. Recall that contrary-to-duty situations encode the problem of what obligations should follow from a normative system in a situation where some of the existing obligations are already being violated. Contrary-to-duty situations were called paradoxical only because SDL failed to give them a reasonable account. They are, in fact, very common in a normative scenario. The norms of a normative system should not be seen as hard constraints, i.e., the obligations in a normative system can be violated and, in those cases, the normative system should also specify what sanctions follow from these violations.

In order to cope with contrary-to-duty paradoxes, IO logic was extended in [20]. In this approach, contrary-to-duty situations were connected with problems related with consistency. The issue was how to deal with excessive output, i.e., those cases in which the output was itself inconsistent or it was inconsistent with respect to the input. In the last case the input is said to be inconsistent with the output and these cases correspond to contrary-to-duty situations.

Using ideas from non-monotonic reasoning, the strategy to overcome this problem was to cut back the set of generators just below the threshold of yielding an excessive output. The following general notions of maxfamily and outfamily were introduced precisely to deal with excessive output.

Given a generating set G and an input A, $maxfamily(G, A)$ is the set of maximal subsets of G for which the output operator yields a set consistent with the input, i.e., the set

$$\{H : H \subseteq G \text{ and } H \text{ is maximal s.t. } out(H, A) \text{ is consistent with A}\}.$$

The set $outfamily(G, A)$ collects the outputs of the elements in $maxfamily(G, A)$:

$$outfamily(G, A) = \{out(H, A) : H \in maxfamily(G, A)\}.$$

Recall that for the operations admitting throughput, namely out_n^+, we have that $A \subseteq out_n^+(G, A)$. Therefore, for those output operators it is equivalent to say that $out_n^+(G, A)$ is consistent with A and that $out_n^+(G, A)$ is itself consistent.

The framework of IO logic allows to define in a straightforward way the important notion of equivalence between generating sets.

Definition 16. *Two generating sets G and G' are equivalent if, for every set of inputs A, we have that $outfamily(G, A) = outfamily(G', A)$.*

5.2 Embedding

We now present an embedding of IO logic in deontic logic programs. The results presented here can be seen as a strengthening of the existing weak connection drawn in [19] between IO logic and Reiter's default logic.

The following lemma shows that, given a generating set G and a set A of formulas, we can define, for each *out* operation, a deontic logic program whose stable model semantics captures the operational semantics given by the respective *out* operator.

Lemma 1. *Let G be a generating set, A an input consistent with the output, and φ and ψ classical propositional formulas.*

1) Let $\mathcal{P}_1 = \{\mathbf{O}(\psi) \leftarrow \varphi : \langle \varphi, \psi \rangle \in G\} \cup$
$\{\varphi \leftarrow : \varphi \in A\}$.

Then, $out_1(G, A) = \{\varphi : \mathbf{O}(\varphi) \in S_{\mathcal{P}_1}\}$.

2) Let $\mathcal{P}_3 = \{\psi \leftarrow \varphi : \langle \varphi, \psi \rangle \in G\} \cup$
$\{\varphi \leftarrow : \varphi \in A\} \cup$
$\{\mathbf{O}(\psi) \leftarrow \varphi : \langle \varphi, \psi \rangle \in G\}$.

Then, $out_3(G, A) = \{\varphi : \mathbf{O}(\varphi) \in S_{\mathcal{P}_3}\}$.

3) Let $\mathcal{P}_1^+ = \{\mathbf{O}(\psi) \leftarrow \varphi : \langle \varphi, \psi \rangle \in G\} \cup$
$\{\varphi \leftarrow : \varphi \in A\} \cup$
$\{\mathbf{O}(\varphi) \leftarrow : \varphi \in A\}$.

Then, $out_1^+(G, A) = \{\varphi : \mathbf{O}(\varphi) \in S_{\mathcal{P}_1^+}\}$.

4) Let $\mathcal{P}_3^+ = \{\mathbf{O}(\psi) \leftarrow \varphi : \langle \varphi, \psi \rangle \in G\} \cup$
$\{\mathbf{O}(\psi) \leftarrow \mathbf{O}(\varphi) : \langle \varphi, \psi \rangle \in G\} \cup$
$\{\mathbf{O}(\varphi) \leftarrow : \varphi \in A\} \cup$
$\{\varphi \leftarrow : \varphi \in A\}$.

Then, $out_3^+(G, A) = \{\varphi : \mathbf{O}(\varphi) \in S_{\mathcal{P}_3^+}\}$.

Note that for the embedding in the lemma, standard normal logic programs are not enough. Not only they do not consider obligations but, equally important, they do not allow for complex formulas in the heads and bodies of rules. Bare in mind that the φ and ψ can be any classical propositional formula. However, in this lemma we only needed to consider definite deontic logic programs, i.e. deontic logic programs without default negation. This is a consequence of the monotonicity of unconstrained IO logic. Contrarily, constrained IO logic has an intrinsic non-monotonic flavor and, as we will see below, default negation plays a fundamental role in the embedding.

In [19] it was showed that, given a generating set G, an input A, and assuming that we take out_3^+ as the *out* operation, there is a relation between the elements of $outfamily(G, A)$ and the default extensions of a Reiter's default system obtained from G and A. In fact, the default extensions are usually a strict subset of $outfamily(G, A)$, corresponding to the maximal elements. The following theorem can be seen as a strengthening of that result, as we prove that, for out_1^+ and out_3^+, we can capture the entire $outfamily$ using our stable models semantics.

Theorem 3. *Let G be a generating set, A an input, and let φ and ψ stand for classical propositional formulas.*

1) Consider the deontic logic program over an extended language that contains a constant $\overline{\mathbf{O}(\psi)}$ for every classical propositional formula $\psi \in L_{CPL}$.

$$\begin{aligned}\mathcal{P}_1 = &\{\mathbf{O}(\psi) \leftarrow \varphi, not\ \overline{\mathbf{O}(\psi)} : \langle \varphi, \psi \rangle \in G\} \cup \\ &\{\varphi \leftarrow : \varphi \in A\} \cup \\ &\{\underline{\mathbf{O}(\varphi)} \leftarrow : \varphi \in A\} \cup \\ &\{\overline{\mathbf{O}(\psi)} \leftarrow not\ \mathbf{O}(\psi) : \psi \in L_{CPL}\}\end{aligned}$$

Then, taking out_1^+ as the out operator, we have

$$outfamily(G, A) = \bigcup_{T \in SM(\mathcal{P}_1)} \{\varphi : \mathbf{O}(\varphi) \in T\}.$$

2) Consider the deontic logic program over an extended language that contains a constant $\overline{b}$ for every classical propositional formula $b \in L_{CPL}$.

$$\begin{aligned}\mathcal{P}_3 = &\{\psi \leftarrow \varphi, not\ \overline{\psi} : \langle \varphi, \psi \rangle \in G\} \cup \\ &\{\varphi \leftarrow : \varphi \in A\} \cup \\ &\{\overline{\psi} \leftarrow not\ \psi : \psi \in L_{CPL}\}\end{aligned}$$

Then, taking out_3^+ as the out operator, we have

$$outfamily(G, A) = SM(\mathcal{P}_3)_{|L_{CPL}}.$$

One could wonder why, in the above theorem, and in the case of out_3^+, we did not need to consider deontic logic programs with deontic operators. The reason is that if we admit throughput, i.e., inputs to be part of the output, and reusability, i.e., outputs can be reused as inputs, the deontic reading of a pair $\langle \varphi, \psi \rangle$ is no longer accurate. This fact, which was already noticed in [21], happens because the reuse of outputs as inputs dilutes the difference between facts and the obligation of these facts. However, note that even though in the case of out_3^+ obligations are not used, standard logic programming is not enough for the above embedding. The reason is that, as already noted above after Lemma 1, we need to consider complex propositional formulas in the body and head of rules, something that is not possible in standard logic programs.

The above embedding theorem shows how we can recast IO logic in deontic logic programming. An interesting question now is what additional features can deontic logic programming immediately bring to IO logic.

First of all, it is very clear that deontic logic programs have a richer language. Note that the deontic logic programs necessary to embed IO logic are not very expressive compared with the expressivity of a normal deontic logic program. Moreover, in deontic logic programming we have an explicit use of default negation, which is fundamental to model exceptions. Also, deontic logic programs can have complex deontic formulas not only in the head, but also in the body of a rule. This is fundamental to model violations of obligations and to specify sanctions in case of violations.

Another fundamental notion that comes for free in the context of deontic logic programming is the notion of equivalence between normative systems. The following corollary, which is an immediate consequence of the embedding theorem, draws a tight connection between equivalence of generating sets in IO logic and the notion of uniform equivalence of deontic logic programs.

Corollary 1. *Suppose that we take* out_3^+ *as the out operator. Then, two generating sets* G *and* G' *are equivalent if their corresponding deontic logic programs* $\mathcal{P}_G = \{\psi \leftarrow \varphi, not\ \overline{\psi} : \langle \varphi, \psi \rangle \in G\} \cup \{\overline{\psi} \leftarrow not\ \psi : \psi \in L_{CPL}\}$ *and* $\mathcal{P}_{G'} = \{\psi \leftarrow \varphi, not\ \overline{\psi} : \langle \varphi, \psi \rangle \in G'\} \cup \{\overline{\psi} \leftarrow not\ \psi : \psi \in L_{CPL}\}$ *are uniformly equivalent.*

As we saw in Section 4, we can check strong equivalence of deontic logic programs using logical equivalence in deontic HT. Interestingly, this immediately gives a sound tool to check equivalence of generating sets.

Corollary 2. *Suppose that we take* out_3^+ *as the out operator. Then, two generating sets* G *and* G' *are equivalent if the sets of* HT_{SDL} *formulas corresponding to the programs* $\mathcal{P}_G = \{\psi \leftarrow \varphi, not\ \overline{\psi} : \langle \varphi, \psi \rangle \in G\} \cup \{\overline{\psi} \leftarrow not\ \psi : \psi \in L_{CPL}\}$ *and* $\mathcal{P}_{G'} = \{\psi \leftarrow \varphi, not\ \overline{\psi} : \langle \varphi, \psi \rangle \in G'\} \cup \{\overline{\psi} \leftarrow not\ \psi : \psi \in L_{CPL}\}$ *are logically equivalent in* HT_{SDL}.

Note that the above test is sound but it is not complete. To be more precise, if the sets of HT_{SDL} formulas corresponding to the programs $\mathcal{P}_G$ and $\mathcal{P}_{G'}$ are not logically equivalent in HT_{SDL}, then we cannot conclude anything about the equivalence of G and G'.

We end this section with the use of deontic logic programs, contrasted with IO logic, in the contrary-to-duty situation of Example 1.

Example 5. Recall the normative statement given in Example 1.

The following is the representation proposed in [21] of that statement using IO logic:

$$\langle \mathbf{t},\ \ \sim (dog \vee fence) \rangle \qquad\qquad \langle dog,\ \ fence \wedge warningSign \rangle$$

The formula $\mathbf{t}$ stands for a tautology. As in our first attempt to model this situation using deontic logic programs, in IO logic the unconstrained output

gives an excessive output whenever *dog* is the case. In fact, the output is not only inconsistent with the input, but it is also itself inconsistent. The output is itself inconsistent because it includes both $fence$ and $\sim fence$, and it is inconsistent with the input because it includes $\sim dog$.

The use of constrained output solves this particular problem. In fact, we have that:

$$\begin{aligned} maxfamily(G, A) &= \{\{\langle dog,\ fence \wedge warningSign\rangle\}\} \\ outfamily(G, A) &= \{Cn(fence \wedge warningSign)\} \end{aligned}$$

Intuitively, since *dog* is the case, the conditional norm $\langle \mathbf{t},\ \sim (dog \vee fence)\rangle$ is always discarded and only the consequences of the other conditional norm are considered. Although in this particular formulation of the example the strategy behind the definitions of $maxfamily$ and $outfamily$ gives a reasonable solution, this is not always the case. As it was pointed out in [21], this strategy is very sensible to how the generating set is written, and in some case it cuts too deeply the output. As an example, suppose that we only consider the first conditional norm $\langle \mathbf{t},\ \sim (dog \vee fence)\rangle$. If *dog* is the case then, surprisingly, $outfamily(G, A)$ only contains the set of tautologies and therefore does not include $\sim fence$.

The motivation behind the idea of constraining the output to deal with contrary-to-duty situations is, as argued in [20], the fact that we should not consider obligations that are already being violated. In the cottage example, we should not conclude that it is obligatory not to have a dog because having a dog is seen as an unchangeable fact. Perhaps this argument is acceptable in the context of IO logic. We argue, however, that this kind of reasoning is not accurate if we want to reason about violation of obligations. In deontic logic programming we want (and can!) reason about the violation of obligations. Consider that, in the cottage example, we have rules for applying sanctions in case of violations, i.e., we augment the above normative system with the rules $\mathbf{O}(fineD) \leftarrow \mathbf{O}(\sim dog), dog$ and $\mathbf{O}(fineF) \leftarrow \mathbf{O}(\sim fence), fence$. Then, given that we have a dog and a fence, the obligation $\mathbf{O}(fineD)$ is entailed by the system but $\mathbf{O}(fineF)$ is not. This kind of reasoning would not be possible if we were assuming that $\mathbf{O}(\sim dog)$ should not follow from the normative system when *dog* is the case.

6 Conclusions and future work

In this paper we have introduced a framework for representing and reasoning about normative systems which combines the expressivity of standard deontic logic with non-monotonic logic programs. We have defined a stable model semantics and studied the problem of equivalence between normative systems. We also proved that an important part of IO logic can be embedded in our framework.

Although we have defined a stable model semantics, the approach used in this paper allows to easily define a well-founded semantics, by making use of logical foundations of the well-founded semantics [5]. As usual, this could be seen as a sound skeptical approximation of the stable model semantics.

Being declarative, our normative framework could easily be integrated in normative multi-agent system that use declarative languages for modeling norms, as for example [30, 15]. This would allow an increasing of expressivity of their norm languages. Although this is not usually the main focus of such systems, it was realized, for example in [31], the need for more expressive declarative languages for representing norms.

Our approach could be easily adapted to combine non-monotonic logic programs with extensions of SDL. Interesting examples include dyadic deontic logics and temporal extensions of SDL. The latter would allow to incorporate some notion of time in the framework.

Another interesting topic is the study of how well-known tools for updating logic programs could be used for the fundamental problem of updating normative systems.

Finally, we would like to implement our framework by combining an answer set solver with a SDL reasoner, in order to automate the reasoning mechanism.

References

1. M. Alberti, A. S. Gomes, R. Gonçalves, M. Knorr, J. Leite, and M. Slota. Normative systems require hybrid knowledge bases. In *Proceedings of the 11th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2012)*, June 2012.
2. M. Alberti, A. S. Gomes, R. Gonçalves, J. Leite, and M. Slota. Normative systems represented as hybrid knowledge bases. In J. Leite, P. Torroni, T. Ågotnes, G. Boella, and L. van der Torre, editors, *CLIMA*, volume 6814 of *Lecture Notes in Computer Science*, pages 330–346. Springer, 2011.
3. G. Boella, G. Governatori, A. Rotolo, and L. van der Torre. A logical understanding of legal interpretation. In F. Lin, U. Sattler, and M. Truszczynski, editors, *KR*. AAAI Press, 2010.
4. G. Brewka. Well-founded semantics for extended logic programs with dynamic preferences. *J. Artif. Intell. Res. (JAIR)*, 4:19–36, 1996.
5. P. Cabalar, S. P. Odintsov, and D. Pearce. Logical foundations of well-founded semantics. In P. Doherty, J. Mylopoulos, and C. A. Welty, editors, *International Conference on Principles of Knowledge Representation and Reasoning – KR*, pages 25–35. AAAI Press, 2006.
6. B. Chellas. *Modal Logic: An Introduction*. Cambridge University Press, 1980.
7. R. Chisholm. Contrary-to-duty imperatives and deontic logic. *Analysis*, 24(2):33–36, 1963.
8. T. Eiter and M. Fink. Uniform equivalence of logic programs under the stable model semantics. In C. Palamidessi, editor, *Logic Programming, 19th International Conference, ICLP 2003*, volume 2916 of *Lecture Notes in Computer Science*, pages 224–238. Springer, 2003.
9. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *ICLP*, pages 1070–1080. MIT Press, 1988.
10. R. Gonçalves and J. J. Alferes. Parametrized logic programming. In T. Janhunen and I. Niemelä, editors, *Logics in Artificial Intelligence – JELIA*, volume 6341 of *LNCS*, pages 182–194. Springer, 2010.

11. R. Gonçalves and J. J. Alferes. Parametrized equilibrium logic. In J. P. Delgrande and W. Faber, editors, *LPNMR*, volume 6645 of *Lecture Notes in Computer Science*, pages 236–241. Springer, 2011.
12. G. Governatori and A. Rotolo. Bio logical agents: Norms, beliefs, intentions in defeasible logic. *Autonomous Agents and Multi-Agent Systems*, 17(1):36–69, 2008.
13. R. Hilpinen. *Deontic Logic: Introductory and Systematic Readings*. Kluwer Boston, 1981.
14. J. F. Horty. Deontic logic as founded on nonmonotonic logic. *Ann. Math. Artif. Intell.*, 9(1-2):69–91, 1993.
15. J. Hubner, J. Sichman, and O. Boissier. Developing organised multiagent systems using the moise+ model: programming issues at the system and agent levels. *Int. J. Agent-Oriented Softw. Eng.*, 1:370–395, 2007.
16. D. Lewis. *Semantic analyses for dyadic deontic logic*. Cambridge University Press, 1999.
17. V. Lifschitz, D. Pearce, and A. Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2:2001, 2000.
18. J. W. Lloyd. *Foundations of Logic Programming*. Springer, 1984.
19. D. Makinson, Leendert, and V. D. Torre. Input-output logics. *Journal of Philosophical Logic*, 29:2000, 2000.
20. D. Makinson and L. van der Torre. Constraints for input/output logics. *Journal of Philosophical Logic*, 30:155–185, 2001.
21. D. Makinson and L. W. N. van der Torre. What is input/output logic? input/output logic, constraints, permissions. In G. Boella, L. W. N. van der Torre, and H. Verhagen, editors, *Normative Multi-agent Systems*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.
22. L. T. McCarty. Defeasible deontic reasoning. *Fundam. Inform.*, 21(1/2):125–148, 1994.
23. D. Nute. *Defeasible deontic logic*. Springer, 1997.
24. D. Pearce. Equilibrium logic. *Ann. Math. Artif. Intell.*, 47(1-2):3–41, 2006.
25. D. Pearce and A. Valverde. Uniform equivalence for equilibrium logic and logic programs. In V. Lifschitz and I. Niemelä, editors, *Logic Programming and Nonmonotonic Reasoning – LPNMR 2004*, volume 2923 of *Lecture Notes in Computer Science*, pages 194–206. Springer, 2004.
26. H. Prakken and G. Sartor. Argument-based extended logic programming with defeasible priorities. *Journal of Applied Non-Classical Logics*, 7(1), 1997.
27. H. Prakken and M. J. Sergot. Contrary-to-duty obligations. *Studia Logica*, 57(1):91–115, 1996.
28. Y. U. Ryu and R. M. Lee. *Defeasible deontic reasoning: a logic programming model*, pages 225–241. John Wiley and Sons Ltd., Chichester, UK, 1993.
29. M. J. Sergot, F. Sadri, R. A. Kowalski, F. Kriwaczek, P. Hammond, and H. T. Cory. The british nationality act as a logic program. *Commun. ACM*, 29:370–386, May 1986.
30. N. A. M. Tinnemeier, M. Dastani, and J.-J. C. Meyer. Roles and norms for programming agent organizations. In C. Sierra, C. Castelfranchi, K. S. Decker, and J. S. Sichman, editors, *AAMAS (1)*, pages 121–128. IFAAMAS, 2009.
31. N. A. M. Tinnemeier, M. Dastani, J.-J. C. Meyer, and L. W. N. van der Torre. Programming normative artifacts with declarative obligations and prohibitions. In *IAT*, pages 145–152. IEEE, 2009.
32. L. W. N. van der Torre. Contextual deontic logic: Normative agents, violations and independence. *Ann. Math. Artif. Intell.*, 37(1-2):33–63, 2003.
33. G. H. von Wright. Deontic logic. *Mind*, 60:1–15, 1951.