CÒIR: Verifying Normative Specifications of Complex Systems

Luca Gasparini$^{1(\boxtimes)}$, Timothy J. Norman^{1}, Martin J. Kollingbaum^{1}, Liang Chen^{1}, and John-Jules C. Meyer^{2}

^{1} Department of Computing Science, University of Aberdeen, Aberdeen, UK
{l.gasparini,t.j.norman,m.j.kollingbaum}@abdn.ac.uk
^{2} Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands
j.j.c.meyer@uu.nl

Abstract. Existing approaches for the verification of normative systems consider limited representations of norms, often neglecting collective imperatives, deadlines and contrary-to-duty obligations. In order to capture the requirements of real-world scenarios, these structures are important. In this paper we propose methods for the specification and formal verification of complex normative systems that include contrary-to-duty, collective and event-driven imperatives with deadlines. We propose an operational syntax and semantics for the specification of such systems. Using Maude and its linear temporal logic model checker, we show how important properties can be verified for such systems, and provide some experimental results for both bounded and unbounded verification.

Keywords: Model checking · Normative systems · Collective imperatives

1 Introduction

The specification and verification of properties of normative systems is an important consideration for the design of complex distributed systems [1,6]. Motivated by the need to capture the requirements of real world scenarios, research on the specification of normative systems has explored conditional [18], event-governed (e.g. activation/expiration condition) norms [16], collective imperatives [9,14], imperatives with deadlines [7], and contrary-to-duty (CTD) norms [18]. A further focus has explored mechanisms for the analysis of systems of norms for the purpose of identifying and resolving conflicts between norms and plans [19]. Although such analyses are of benefit, for safety critical systems it is important to analyse the interactions between normative constraints and agents' actions as a system evolves. For these reasons the use of model checking [3] techniques to analyse liveness and safety properties of norm-governed systems has been explored [1,6,8]. To date, however, this research has focussed on restricted representations of norms such as labelling states or transitions as compliant/non-compliant. Ågotnes et al. [1], for example, study the complexity of this model checking problem for different robustness-related properties; e.g. whether a certain property is guaranteed in the event of a subset of agents violating a norm.

The focus of this paper is on how to efficiently apply model checking to analyse properties of normative systems specifications with richer representations of norms. In particular, we consider event-governed conditional norms, deadlines for the fulfilment of obligations, and contrary to duty and group imperatives. The contributions we claim are as follows: (i) We propose a norm specification language that is sufficiently expressive to capture all the features discussed above, namely CÒIR [1]; (ii) a Structural Operational Semantics (SOS) [15] for a monitoring component that, given a description of the environment, keeps track of activation, expiration, fulfilment, and violations of norms; and (iii) a realisation of this component using the Maude [4] rewriting logic framework, which allows us to perform formal analysis of normative systems specifications. A particular challenge is that representing time explicitly (in order to reason about temporal deadlines) makes the problem undecidable. For these reasons we explore both the use of bounded model checking and model abstraction to obtain a finite Kripke structure for unbounded model checking. We present some results of both these approaches in an example domain that motivates the requirements for us considering such a rich representation of norms.

2 Motivating Example

Consider a coalition of agents of the sea-guard, consisting of a set of *Unmanned aerial vehicles (UAVs)*, *helicopters*, and *boats*. Their goal is to monitor and intercept unauthorized boats trying to access a restricted area. The norms that guide the behaviour of the coalition are: (1) At any moment at least one member of the coalition must monitor the area. Moreover, we prefer having UAVs monitoring the area over helicopters. We assume that only helicopters and UAVs are capable of monitoring. (2) Whenever an unauthorized boat enters the area, a member of the coalition must intercept it before a certain deadline expires. (3) If no one intercepts the boat, then at least one member of the coalition must send a report to head-quarters before a certain deadline expires. These are all examples of collective imperatives: they require *at least one member* of the coalition to act. Norm 3 is also a CTD obligation that is activated in the event of a violation of the obligation 2. Moreover, norms 2 and 3 require the agents to perform an action *before a certain deadline* (a liveness property), while norm 1 requires that *at any given moment* someone is monitoring the area (a safety property).

3 CÒIR Norm Specification

We now introduce a formalism for representing norms that satisfies our requirements, which we call CÒIR. We allow for the definition of *obligations with deadlines* and *prohibitions* and we assume that everything that is not prohibited is

[1] CÒIR is Scottish Gaelic for obligation.

permitted. Compliance with norms is evaluated against a knowledge base *KB* that is dynamically updated to represent the environment and the observable properties of the agents acting within it. We rely on the closed-world assumption, which we believe to be reasonable in a verification setting. We include the description of previous violations in the knowledge base. These can then be used to activate CTD norms. An issue that has been discussed, for example, by Dignum et al. [7] is whether an obligation with a deadline should persist or be deactivated after a violation; i.e. after the deadline has expired without the obligation being fulfilled. CÒIR supports the specification of either of these alternatives. By default obligations do not expire when violated, but, thanks to the fact that violations are represented in *KB*, it is always possible to specify the expiration condition as being triggered by a violation of the current instance.

3.1 Syntax

A norm `nd`i is defined as a tuple $\langle id_i, mod_i, act_i, exp_i, goal_i, ddl_i \rangle$ where: id_i is a unique identifier; $mod_i \in \{O, F\}$ specifies whether the norm is an obligation with deadline or a prohibition; act_i (activation condition) describes a pattern that, when matched in *KB*, causes a norm instance to be detached; $goal_i$ represents the situation that needs to be brought about (for an obligation) or avoided (for a prohibition); exp_i (expiration condition) is a condition that, when met, causes the expiration of the instance; and the deadline for the fulfilment of the norm (ddl_i) can be temporal or symbolic and is defined only for obligations.

Figure 1 shows the EBNF grammar of the operational language used to represent the components of a norm specification. `functor` and `strTerm` are identified by strings that start with a letter, `numTerm` by numbers and `varTerm` by strings that start with a `?` character. `?actTime`, `?violTime`, `?tick`, `?this-id`, `?violated` and `?flag` are reserved terms. The description of the environment, *KB*, consists of a set of ground predicates; i.e. predicates with no `varTerm`. Intuitively, a `boolExpr` represents a condition that is evaluated against *KB* returning a boolean result, while a `formula` is a pattern with a set of variables that is evaluated by returning the set of substitutions that make the pattern match a subset of *KB*. In a norm description, act_i is represented by a `formula`, while exp_i, $goal_i$, and ddl_i are `boolExpr`s.

The formula `VIOLATION-OF(`n, s`)` is matched when there is a violation of norm n and is used for the activation of CTD obligations. The meaning of the parameter s will be explained in Sect. 4. The meanings of `EQUALS`, `EXISTS` and the usual boolean operators are intuitive. `TEMPORAL(`n`)` is evaluated to `true` if a temporal deadline has expired, while `VIOLATED` can be used in exp_i and returns `true` if the instance being evaluated has been violated. `COUNT (`v `IN` $\{f\}$`) > n` evaluates to `true` if the number of different assignments of the variable v that matches the pattern f is higher than the number n.

```
⟨constTerm⟩      = ⟨strTerm⟩ | ⟨numTerm⟩ ;
⟨term⟩           = ⟨constTerm⟩ | ⟨varTerm⟩ | ⟨predicate⟩ ;
⟨predicate⟩      = ⟨functor⟩ "(" ⟨term⟩ { "," ⟨term⟩ } ")" ;
⟨binding⟩        = "BIND( " ⟨varTerm⟩ "," ⟨term⟩ ")" ;
⟨violationCond⟩ = "VIOLATION-OF(" ⟨numTerm⟩ "," ⟨strTerm⟩ ")" ;
⟨formula⟩        = ⟨predicate⟩ | ⟨binding⟩ | ⟨violationCond⟩ | ⟨formula⟩ "/\" ⟨formula⟩
                 | ⟨formula⟩ "\/" ⟨formula⟩  | "IN {" ⟨formula⟩ "} FILTER" ⟨boolExpr⟩ ;
⟨constant⟩       = "false" | "true" ;
⟨existsPattern⟩ = "EXISTS {" ⟨formula⟩ "}" ;
⟨equalsCond⟩     = "EQUALS (" ⟨term⟩ "," ⟨term⟩ ")" ;
⟨temporal⟩       = "TEMPORAL (" ⟨numTerm⟩ ")" ;
⟨violated⟩       = "VIOLATED" ;
⟨compareOp⟩      = "=" | ">" | "<" | "<=" | ">=" ;
⟨count⟩          = "COUNT(" ⟨varTerm⟩ "IN{" ⟨formula⟩ "})" ⟨compareOp⟩⟨numTerm⟩ ;
⟨boolExpr⟩       = "NOT" ⟨boolExpr⟩ | ⟨violated⟩ |  ⟨count⟩ | ⟨boolExpr⟩ "/\" ⟨boolExpr⟩
                 | ⟨boolExpr⟩ "\/" ⟨boolExpr⟩ | ⟨existsPattern⟩ | ⟨equalsCond⟩ | ⟨constant⟩ ;
```

Fig. 1. EBNF grammar for the COIR language.

3.2 Representing Collective Obligations

We now discuss how our formalism allows us to represent different types of collective obligations [14]. In contrast to Tinnemeier et al. [17], we allow $goal_i$, exp_i, and ddl_i to include variables that have not been bound at activation time. Through the use of the patterns $\texttt{EXISTS}\{f_i\}$ and $\texttt{NOT EXISTS}\{f_i\}$ we are able to express existential and universal quantification on these variables. Inspired by Norman and Reed [14] we discuss some common patterns of collective obligations and show how they can be expressed in our language (See [9,14] for discussions of responsibility in collective obligations). In order to ease the presentation, we assume that agents are organized in groups, group membership is represented by predicates of the type `memberOf(agent,group)`, and an agent's performance of an action by `perform(agent,action)`.

Joint distributive obligations are obligations where all the members of group `g` are responsible for all the members of the group performing the action `a`. This can be expressed by an obligation where:

$$\begin{aligned} act_i &= \texttt{memberOf(?add,g)} \\ goal_i &= \texttt{NOT EXISTS } \{\texttt{IN } \{\texttt{memberOf(?ag,g)}\} \\ &\qquad \texttt{FILTER NOT EXISTS } \{\texttt{perform(?ag,a)}\}\} \end{aligned}$$

$goal_i$ is met when there is no member of `g` that has not performed `a`; i.e. when all the members of `g` have performed `a`. As a result, if any of the members of the group do not perform the task, all the members will be responsible for the violation. Alternatively we could consider the group as an entity to be responsible for the fulfilment of the obligation by specifying the activation condition as:

$$act_i = \texttt{BIND(?add,g)}$$

and referring to the group as `?add` in the goal. Note that if a group has no members, such an obligation would be trivially fulfilled. It might be appropriate to add the constraint `EXISTS{memberOf(?add,g)}` in act_i or in $goal_i$.

Joint collective obligations specify that all the members of a group `g` are responsible for at least one member of the group performing the action `a`.

$$\begin{aligned} act_i &= \texttt{memberOf(?add,g)} \\ goal_i &= \texttt{EXISTS\{ memberOf(?ag,g) /\textbackslash\ perform(?ag,a)\}} \end{aligned}$$

4 CÒIR Semantics

We define the semantics of CÒIR through a Structural Operational Semantics (SOS) [15], a framework for the description of the semantics of programming and specification languages. SOS consists of a set of transition rules that generate a transition system whose states are called configurations. Transition rules are of the form $\frac{P}{C \to C'}$ meaning that, whenever P holds, a transition from the configuration C to C' is applicable. We use SOS to describe how the active norm instances and violations are updated every time we detect a change in KB.

In formalising these semantics we assume two functions that evaluate `formula` and `boolExpr`; these will be summarised below. We define a substitution $\theta_j \in \Theta$ as a set of assignments $[v/c]$ where c is a `constTerm` and v a `varTerm`. Formulae are evaluated by means of a function $match : 2^P \times Q \to 2^\Theta$, where P is the set of all predicates, Q the set of all formulae, and Θ the set of all substitutions. Intuitively, $match(KB, f)$ returns all the substitutions θ_i such that $f \cdot \theta_i$ is entailed by KB. Boolean expressions (`boolExpr`) are evaluated by means of a function $eval : 2^P \times E \times \Theta \to bool$ where E is the set of all `boolExpr` and $bool \in \{$ `true`, `false` $\}$. A norm instance $[id_i, \theta_j, at]$ is detached at time at for each substitution $\theta_j \in match(KB, act_i)$. Then $eval(KB, e, \theta_j)$ is used to evaluate exp_i, $goal_i$, and ddl_i. The addressee of the norm, identified by the value assigned to `?add` in θ_j, is responsible for complying with the obligation (reaching a state where $eval(KB, goal_i, \theta_j) =$ `true` before the deadline) or with the prohibition (avoiding states where $eval(KB, goal_i, \theta_j) =$ `true` until the prohibition expires).

A further issue to address prior to detailing the transition rules of our operational semantics is that of "duplicate activations". Consider a simplified version of norm 3 from Sect. 2. We specify its activation condition as follows:

```
type(?add,coalition) /\ type(?boat,unBoat)
  /\ type(?area,rArea) /\ inArea(?boat,?area)
```

In other words, an instance of the obligation to send a report should be detached when an unauthorized boat is in the restricted area. Intuitively, if the same boat remains in the restricted area for more than one consecutive instant of time, we do not want the coalition members to send more than one report. However, if the boat exits and then re-enters the area, we would expect the coalition to be obliged to send another report. Formally, if we denote by KB_t the state of the knowledge base at time t, we capture this distinction by activating an instance of a norm `ndi`, associated with a substitution θ_j, at an instant of time t whenever $\theta_j \in match(KB_t, act_i)$ and $\theta_j \notin match(KB_{t-1}, act_i)$; i.e. when we find a substitution such that act_i goes from "unmatched" to "matched" in two subsequent instants of time. To do that we keep record of the instances $[id_i, \theta_j, at]$ such that the act_i was matched in the previous instant of time.

Following Dennis et al. [6], in order to enforce an order of execution among the transitions of the operational semantics, we organize the reasoning cycle in three stages: (**A**) Deactivate instances for which the expiration condition holds or the obligation has been fulfilled; (**B**) Check for violations of active obligations (if the deadline has passed, but the goal has not been achieved) and prohibitions (if the state to avoid is achieved). (**C**) Check for the activation of new norms and update the list of previously matched instances.

In the following we denote by $a_1 : a_2 : \ldots$ a list of elements and we use ϵ to indicate the end of a list. Moreover, we assume that KB contains a predicate `cT(n)`, where `n` is a `numTerm` that represents the current time of the system and we denote by $time(KB)$ the value `n` such that `cT(n)` $\in KB$. A configuration $Conf$ is defined as $\langle KB, \Delta, I, \Pi, \Phi, \Sigma, r\rangle$ where KB is the current state of the knowledge base, Δ is a list of norm descriptions, I is the list of active norm instances and Π the list of previously matched instances, which, as discussed above, is needed to avoid the problem of multiple activations. Φ is the set of violations detected in the current reasoning cycle[2], and a violation is represented as $v = [id_i, \theta, t]$, where t corresponds to the violation time. Σ is the stage of the computation and r is a flag that is set initially to `false`, and changed to `true` if we need to loop again through the reasoning cycle. This is necessary because, whenever we activate a new instance (stage **C**), we need to check whether this is instantly fulfilled or violated (**A** and **B**). Moreover, detecting a violation (**B**) could trigger an expiration or an activation (**A** and **C**).

The initial configuration is $\langle KB_0, \Delta, \epsilon, \epsilon, \epsilon, \mathbf{A}, \texttt{false}\rangle$, where KB_0 describes the initial state and Δ the normative specification. We now illustrate the key rules of the operational semantics. For each rule we include only the components of the configuration that are involved in it.

Rule R1 applies when the first instance in I is such that its expiration condition holds. In this case we simply remove the instance from the list. Similarly another rule (not included) is defined for the case of a fulfilled obligation. Rule R2 accounts for the case where the first instance in the list is a prohibition and the expiration condition is not met. In this case we move the instance to the end of the list, after the ϵ symbol. We write a similar rule (not included) for an obligation instance that it is neither fulfilled nor expired. Rule R3 represents the end of stage A, which occurs when the first instance is ϵ.

[2] We refer to the whole updating procedure as a reasoning cycle, while **A**, **B** and **C** are the stages of a cycle.

$$\frac{\langle KB, \Delta, [id_i, \theta_j, at] : I, \mathbf{A}\rangle,\ nd_i \in \Delta,\ eval(KB, exp_i, \theta_j) = \texttt{true}}{\langle KB, \Delta, [id_i, \theta_j, at] : I, \mathbf{A}\rangle \rightarrow \langle KB, \Delta, I, \mathbf{A}\rangle} \tag{R1}$$

$$\frac{nd_i \in \Delta,\ mod_i = F,\ eval(KB, exp_i, \theta_j) = \texttt{false}}{\langle KB, \Delta, [id_i, \theta_j, at] : I, \mathbf{A}\rangle \rightarrow \langle KB, \Delta, I : [id_i, \theta_j, at], \mathbf{A}\rangle} \tag{R2}$$

$$\frac{\texttt{true}}{\langle \epsilon : I, \mathbf{A}\rangle \rightarrow \langle I : \epsilon, \mathbf{B}\rangle} \tag{R3}$$

Rule R4 detects violated obligations; i.e. obligations whose deadline has expired before the goal is satisfied. Since fulfilled obligations have been deleted in stage **A**, we just need to check whether the deadline has expired. When we detect a violation we update the violations list, add the violation description (denoted by $d([id_i, \theta_j, \tau])$) to KB and we set the flag r to `true` since the violation predicate might trigger the expiration condition of that instance. $d([id_i, \theta_j, \tau])$ consists of a predicate $\texttt{v(}id_i, \texttt{p(}\theta_j\texttt{)}, \tau\texttt{)}$ where $\texttt{p(}\theta_j\texttt{)}$ is a representation of the substitution in the form of a predicate. In rule R5 if the first obligation in the list is not violated we move it at the end of the list. Similarly we add two rules (not included) for prohibitions, where we consider a prohibition to be violated if its goal condition evaluates to `true`. Φ is included to avoid infinite loops. In fact, since rule R4 sets r to `true`, detecting the same violation in each loop would cause infinite iteration. Rule R6, together with the condition $[id_i, \theta_j, \tau] \notin \Phi$ of rule R4 ensures that each violation is detected only once for each reasoning cycle. Another rule similar to rule R3 (not included) is defined for the end of stage **B**.

$$\frac{mod_i = O,\ [id_i, \theta_j, \tau] \notin \Phi,\ eval(KB, ddl_i, \theta_j) = \texttt{true},\ KB^* = KB \cup d([id_i, \theta_j, \tau])}{\langle KB, \Delta, [id_i, \theta_j, at] : I, \Phi, \mathbf{B}, r\rangle \rightarrow \langle KB^*, \Delta, I : [id_i, \theta_j, at], [id_i, \theta_j, \tau] : \Phi, \mathbf{B}, \texttt{true}\rangle} \tag{R4}$$

$$\frac{nd.mod = O, eval(KB, ddl_i, \theta_j) = \texttt{false}}{\langle KB, \Delta, [id_i, \theta_j, at] : I, \mathbf{B}\rangle \rightarrow \langle KB, \Delta, I : [id_i, \theta_j, at], \mathbf{B}\rangle} \tag{R5}$$

$$\frac{[id_i, \theta_j, \tau] \in \Phi}{\langle [id_i, \theta_j, at] : I, \Phi, \mathbf{B}\rangle \rightarrow \langle I : [id_i, \theta_j, at], \Phi, \mathbf{B}\rangle} \tag{R6}$$

Rule R7 checks for the activation of new instances of the first norm nd_i in Δ. Let $\tau = time(KB)$, for each $\theta_j \in match(KB, act_i)$, we add a new instance $[id_i, \theta_j, \tau]$ at the end of Π (list Π_2), while we add to I only those instances that are not in Π (list I_2). The substitutions of the instances added to I_2 are integrated with the assignment of the variables `?actTime` and `?this-id` which are needed to evaluate the `TEMPORAL` and the `VIOLATED` conditions as we will show below. If we activate at least one new instance we set $r = \texttt{true}$. By adding new instances at the end of Π, we ensure that, at the end of the reasoning process, the instances added to Π during the current reasoning cycle will be those after ϵ. Formally the pattern $\Pi_3 : \epsilon : \Pi_4$ identifies with Π_4 all the instances added in the current step and with Π_3 all the instances added during the previous reasoning cycle. This is exploited in rule R8, where, at the end of stage **C**, if r is equal to `false`, we end the reasoning cycle (stage **end**) and discard Π_3 and Φ. We define another rule (not included) for the case where r is equal to `true`. In this case we move ϵ at the end of Δ and go back to stage **A**. In rule R7, when we check if a new instance is not in Π, we consider also instances added in previous loops of the current reasoning cycle. In this way it is guaranteed that we do not reactivate the same instances in each loop.

$$\frac{\begin{array}{c}\theta_k = [\texttt{?actTime}/\tau] \cup [\texttt{?this-id}/id_i] \text{ and} \\ I_2 = \langle [id_i, (\theta_j \cup \theta_k), \tau] : \ldots \rangle \ s.t.\ \theta_j \in match(KB, act_i) \text{ and} \\ eval(KB, exp_i, \theta_j) = \texttt{false} \text{ and } [id_i, \theta_j, \tau - 1] \notin \Pi, \\ \Pi_2 = \langle [id_i, \theta_j, \tau] : \ldots \rangle \ s.t.\ \theta_j \in match(KB, act_i), \\ r* = \texttt{true} \text{ iff } (I_2 \neq \emptyset) \text{ or } (r = \texttt{true})\end{array}}{\begin{array}{c}\langle KB, nd_i : \Delta, I, \Pi, \mathbf{C}, r \rangle \rightarrow \\ \langle KB, \Delta : nd_i, I_2 : I, \Pi : \Pi_2, \mathbf{C}, r* \rangle\end{array}} \quad \text{(R7)}$$

$$\frac{\texttt{true}}{\langle \epsilon : \Delta, \Pi_3 : \epsilon : \Pi_4, \Phi, \mathbf{C}, \texttt{false} \rangle \rightarrow \langle \Delta : \epsilon, \Pi_4 : \epsilon, \epsilon, \mathbf{end}, \texttt{false} \rangle} \quad \text{(R8)}$$

With these transition rules in place, we now provide further details of the *match* and *eval* functions for querying *KB*. We denote by $\theta_i[v]$ the value c assigned by θ_i to the variable v. Given a formula f, $f \cdot \theta_i$ denotes the `formula` obtained by substituting, for each `varTerm` v with an assignment in θ_i, each occurrence of v in f with $\theta_i[v]$. Moreover we say that two substitutions θ_1 and θ_2 are compatible if and only if there is no variable v that is bound in both the substitutions such that its assigned values are different. Formally:

$$compatible(\theta_1, \theta_2) = \texttt{true} \text{ iff } \nexists\, v,\ ([v/c_1] \in \theta_1 \text{ and } [v/c_2] \in \theta_2 \text{ and } c_2 \neq c_1)$$

Let p denote a `predicate`, e a `boolExpr`, f_i a `formula`, v_i a `varTerm`, n and a `numTerm`, s_i a `strTerm` and t a `constTerm`. We denote by $s_1.\theta_k$ the substitution obtained by adding the string s_1 as a prefix to all `varTerms` in θ_k. Figure 2 summarizes the semantics of *match* and *eval*.

The construct `TEMPORAL(n)`, where `n` is a `numTerm`, will be used to evaluate a temporal deadline of `n` steps relative to the activation time of a norm instance. In defining its semantics we assume that the variable `?actTime` is bound in θ_j to the activation time (see Rule R7 of above). The construct `VIOLATION-OF(`n, s`)` presented in Sect. 3.1, can be used in the activation condition of a CTD norm to return the description of a detected violation of a norm with id n. For a violation $[id_j, \theta_j]$, with $n = id_j$, it returns the substitution obtained by adding the prefix s to all the variable names of θ_j. The prefix is added in order to allow the norm designer to distinguish between variables bound by the substitution of the violation and variables bound by the activation condition, even when they have

$$match(KB, p) = \{\theta_i : p \cdot \theta_i \in KB\}$$
$$match(KB, \texttt{BIND (} v_1 \texttt{ , } t \texttt{)}) = \{[v_1/t]\}$$
$$match(KB, f_1 \texttt{ /\textbackslash } f_2) = \{(\theta_1 \cup \theta_2) : \theta_1 \in match(KB, f_1) \text{ and } \theta_2 \in match(KB, f_2) \text{ and } compatible(\theta_1, \theta_2)\}$$
$$match(KB, f_1 \texttt{ \textbackslash/ } f_2) = match(KB, f_1) \cup match(KB, f_2)$$
$$match(KB, \texttt{VIOLATION-OF}(t_1, s_1)) = \{s_1.(\theta_j \cup [\texttt{?violTime}/vt]) : d([t_1, \theta_j, vt]) \in KB\}$$
$$eval(KB, \texttt{EXISTS \{ } f_1 \texttt{ \}}, \theta_i) = \texttt{false} \text{ if } match(KB, f_1 \cdot \theta_i) = \emptyset;\ \texttt{true} \text{ otherwise.}$$
$eval(KB, \texttt{EQUALS (} t_1 \texttt{ , } t_2 \texttt{)}, \theta_i) = \texttt{true}$ iff $t_1 = t_2$. If t_1 or t_2 are varTerm, θ_i is used to replace them with their assigned constant terms.

$eval(KB, \texttt{VIOLATED}, \theta_i) = \texttt{true}$ iff there exists a vt such that $d([t_1, \theta_i, vt]) \in KB$ and $\theta_i[\texttt{?thisId}] = t_1$

$$eval(KB, \texttt{TEMPORAL}(n), \theta_i) = \texttt{true} \text{ iff } \theta_i[\texttt{?actTime}] + n <= time(KB)$$
$eval(KB, \texttt{COUNT}(v_1 \texttt{ IN}\{f_1\}) > n, \theta_i) = \texttt{true}$ iff $|\{\theta_j[v_1] : \theta_j \in match(KB, f_1 \cdot \theta_i)\}| > n$. same for the other compareOp.

NOT, \/, and /\ have the usual meaning when applied to boolExpr

$$match(KB, \texttt{IN \{ } f_1 \texttt{\} FILTER } e) = \{\theta_i : \theta_i \in match(KB, f_1) \text{ and } eval(KB, e, \theta_i) = \texttt{true}\}.$$

Fig. 2. Semantics of *match* and *eval*

the same variable name. The construct VIOLATED is used when we want to ask whether the current instance has been violated (e.g. for the expiration condition of an obligation). It is evaluated to true if *KB* contains the description of a violation of the instance being evaluated. Note that, since, for each instance, we bind the activation time in the substitution, VIOLATED is able to distinguish between violations of different instances associated with the same pair $(\texttt{nd}j, \theta_j)$.

Figure 3 illustrates the life-cycle of an obligation (left) and a prohibition (right) instance in CÒIR. Circles represent states and arrows represent transitions and are labeled with the condition that triggers the transition. A norm instance is activated when the activation condition (act) holds and an equivalent instance (an instance of the same norm associated with the same substitution) is not in the previous matches (Π) list. An active obligation becomes fulfilled when the goal (goal) condition holds, it expires if the expiration condition (exp) but not the goal holds, and it becomes violated if the deadline (ddl) condition holds true before the expiration or the goal condition. Once an obligation instance is violated, it remains so until the expiration condition holds (in which case it becomes expired) or the goal condition holds (in which case it becomes fulfilled). Once an obligation is fulfilled or expired it will remain so for the remainder of the execution. An active prohibition expires when the expiration condition holds, and becomes violated if the goal holds, but the expiration condition does not. A violated prohibition becomes expired if the expiration condition holds. It is important to notice that, when a previously violated norm becomes expired it will not be detected as a current violation. A norm designer, however, can specify the clause NOT VIOLATED in the expiration condition in order to avoid this. The same applies to violated obligations that becomes fulfilled.

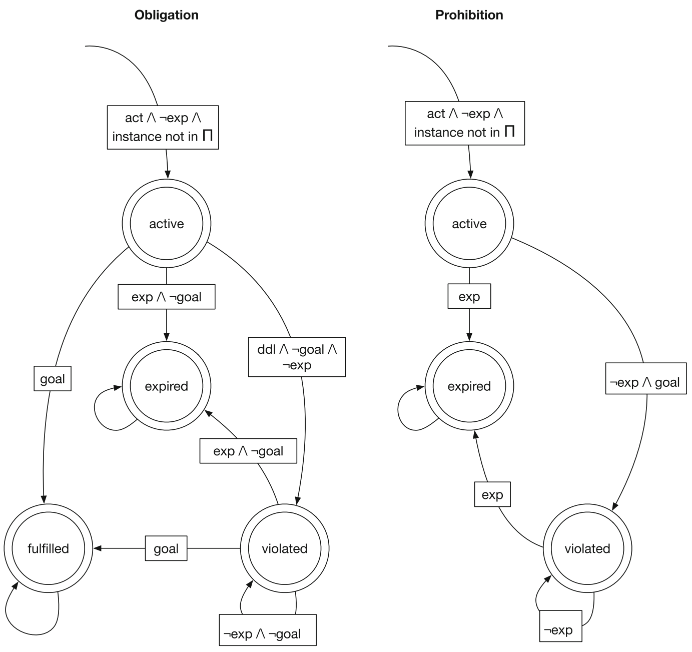

Fig. 3. Norm instance life-cycle

5 The *Seaguard* Example

We now show how we can capture the norms described in our motivating example (Sect. 2) using the CÒIR formalism. Norm 1 states that at any instant of time, at least one agent must monitor the area. This may be captured by a prohibition from achieving a state where no agent is monitoring the area (a safety property). The fact that a UAV monitoring the area is preferred to a helicopter can be represented by separating the norm in two as shown in Fig. 4 (`nd1` and `nd4`). Norm `nd1` is a prohibition that is violated if no UAV is monitoring the area. Norm `nd4` is violated if neither a UAV nor a helicopter is monitoring the area. Therefore, a situation where a UAV is monitoring the area would comply with both the norms, while having a helicopter monitoring would violate only `nd1`.

Norms `nd2` and `nd3` capture the specification of norms 2 and 3 from our motivating example respectively. An instance of the obligation `nd2` is activated, for a coalition, every time an unauthorized boat `?ag1` enters the restricted area `?ar`. The obligation is fulfilled if one member of the coalition `?ag2` intercepts `?ag1` before a deadline of three time steps, while it expires if `?ag1` exits `?ar` or the obligation is violated. Obligation `nd3` is activated by a violation of norm `nd2`, and is addressed to the same coalition. It requires at least one member of the coalition to report the unauthorized access.

```
nd1 = ⟨1, F, act1, false, goal1, false ⟩
   act1 = type(?add,coalition) /\ type(?ar,rArea)
   goal1 = NOT EXISTS{ memberOf(?ag1,?add) /\ type(?ag1,uav) /\ monitoring(?ag1,?ar) }
nd4 = ⟨4, F, act1, false, goal4, false ⟩
   goal4 = NOT EXISTS{ memberOf(?ag1,?add) /\ monitoring(?ag1,?ar) /\
      ( type(?ag1,uav) \/ type(?ag1,heli) ) }
nd2 = ⟨2, O, act2, exp2, goal2,  ddl2⟩
   act2 = IN{ type(?add,coalition) /\ type(?ar,rArea) /\ inArea(?ag1,?ar) } FILTER
      NOT EXISTS{ type(?ag1,?type) /\ subType(?type,authAgent) }
   exp2 = VIOLATED \/ NOT EXISTS { inArea(?ag1,?ar) }
   goal2 = EXISTS{ intercepting(?ag2,?ag1) /\ memberOf(?ag2,?add) }
   ddl2  = TEMPORAL(3)
nd3 = ⟨3, O, act3, exp3, goal3,  TEMPORAL(3)  ⟩
   act3 = IN { type(?add,coalition) /\ VIOLATION-OF(2,v) } FILTER EQUALS(?add,?v:add)
   exp3 = NOT EXISTS{ inArea(?v:ag1,?v:ar) } \/ EXISTS { intercepting(?ag2,?v:ag1) }
   goal3 = EXISTS{ reporting(?ag2,?v:ag1) /\ memberOf(?ag2,?add) }
```

Fig. 4. Specification of norms `nd1`, `nd2` and `nd3`.

6 Formal Verification

In this section we explore the problem of verifying properties of multi-agent systems specified using CÒIR. Firstly we discuss our implementation of the operational semantics in Maude [4], a rewriting logic framework that allows us to specify the semantics of a system by means of rewriting rules. We chose Maude because its syntax for specifying rewriting rules is very close to that for SOS. Moreover, by implementing our system in Maude, we obtain a specification which is executable and on which we can perform formal verification using the Maude Linear Temporal Logic (LTL) model checker. In this way we can: (i) Validate our normative specification; for example by verifying that a specified non compliant behaviour always results in a detected violation; and (ii) Verify how robust a multi-agent system is to violations; for example by verifying if a certain property is guaranteed under certain compliance assumptions [1,8].

We discuss the reasons why, by representing our model as explained in Sect. 4, we obtain an infinite state model. We show how we can use the LTL model checker to perform bounded model checking of the infinite state system, and then show how we can modify our model in order to make the state space finite and apply unbounded model checking.

6.1 Maude Implementation

Maude modules can contain *conditional equations*: simplification rules used to define data-types and language constructs and to specify how they are evaluated by the system. Modules may also contain *conditional rewriting rules*: transition rules that describe how the state of a system can evolve over time. We defined the CÒIR language (Fig. 1) and we implemented the *match* and *eval* functions. We then implemented our operational semantics by means of an operator `reason` that takes as arguments a configuration and returns the configuration resulting from the application of the reasoning cycle. The reasoning process is described by a set of conditional equations, which are a direct (syntactical) translation of the rules of Sect. 4 into the Maude syntax. The dynamics of the system is specified by a set of rules that follow the pattern:

```
crl C => reason( tick(C', n) ) if condition.
```

where `C` and `C'` are two configurations and the only component that can change from `C` to `C'` is the knowledge base. `tick` is a function that takes a configuration `C` and an integer `n` as parameters and increases the time in `C` by `n` units. The meaning of this rule pattern is that, at each step, after applying the changes in the description of the environment, we invoke the `reason` operator to update the list of active instances, previous matches and violations accordingly. The Maude model checker, given one initial state i, and a set of transition rules T, generates a Kripke structure containing all the states that are reachable from i.

6.2 Bounded Model Checking

Properties of a norm-governed multi-agent system can be verified using the Maude LTL model checker. In order to do so we need to define a labelling function λ, specifying the set of atomic propositions $q \in Q$ that hold in some state $s \in S$ [4, Chap. 13]. We denote by $((s \models_\lambda q) = \texttt{true})$ the fact q holds in s and by $((s \models_\lambda q) = \texttt{false})$ the fact that q does not hold in s. The state of a multi-agent system is represented by the configuration *Conf* of the monitoring component. Let Q be the set of all `predicates` as defined in Fig. 1. Equations 1–4 defines λ.

$$\langle KB, \Delta, I, \Pi, \Phi, \Sigma, r\rangle \models_\lambda p = \texttt{true} \text{ if } p \in KB. \tag{1}$$

$$\begin{gathered}\langle KB, \Delta, I, \Pi, \Phi, \Sigma, r\rangle \models_\lambda \mathit{violated}(n) = \texttt{true} \\ \text{if } \exists\, \theta_j, \tau \text{ s.t. : } d([n, \theta_j, \tau]) \in KB\end{gathered} \tag{2}$$

$$\begin{gathered}\langle KB, \Delta, I, \Pi, \Phi, \Sigma, r\rangle \models_\lambda \mathit{violated}(n, t) = \texttt{true} \text{ if} \\ \exists\, \theta_j, \tau \text{ s.t. : } d([n, \theta_j \cup [\texttt{?add/t}], \tau]) \in KB\end{gathered} \tag{3}$$

$$\langle KB, \Delta, I, \Pi, \Phi, \Sigma, r\rangle \models_\lambda p = \texttt{false} \text{ otherwise.} \tag{4}$$

Equation 1 makes it possible to use the predicates of KB as atoms of LTL properties. Equations 2 and 3 define properties about the normative state of a configuration, allowing us to query the model checker for states where a certain norm has been violated (optionally specifying an addressee).

The principal requirement to make the LTL model-checking decidable is for the transition system to have a finite number of reachable states. However, the fact that we represent time explicitly in KB means that the state space is infinite. One way of dealing with this is to limit the state space to the states reachable in a fixed number of transitions, l. We can do this, for example, by modifying the specification of the system so that all the conditional rewriting rules that increase the time by n are applicable only to states where $\mathit{time}(KB) < l - n$. Ideally, however, we want to be able to verify system properties in the unbounded case.

6.3 Unbounded Model Checking

In order to make the unbounded model checking problem decidable, we need to remove any explicit reference to the current time from the semantics. We remove the predicate `cT(n)` from KB and the references to activation and violation time from instances and violations respectively (now represented as $[id_j, \theta_k]$). In order to represent temporal deadlines, we take an approach similar to the one proposed by Lamport [12]. When we activate an instance (Rule R7), instead of binding `?actTime`, we add the assignment $[\texttt{?tick}/\texttt{n}]$ in the substitution of instances of norms that include a statement of type `TEMPORAL(n)`. Rule R7 is substituted with:

$$\frac{\begin{array}{c} I_2 = \langle [id_i, (\theta_j \cup \theta_k)] : \ldots \rangle \ s.t.\ \theta_j \in match(KB, act_i) \text{ and} \\ eval(KB, exp_i, \theta_j) = \texttt{false} \text{ and } \theta_k = isTemp(ddl_i) \text{ and} \\ [id_i, \theta_j] \notin \Pi, \Pi_2 = \langle [id_i, \theta_j] : \ldots \rangle \ s.t.\ \theta_j \in match(KB, act_i), \\ r* = \texttt{true} \text{ iff } (I_2 \neq \emptyset) \text{ or } (r = \texttt{true}) \end{array}}{\langle KB, nd_i : \Delta, I, \Pi, \mathbf{C}, r \rangle \rightarrow \langle KB, \Delta : nd_i, I_2 : I, \Pi : \Pi_2, \mathbf{C}, r* \rangle} \tag{R7*}$$

where $isTemp(ddl_i)$ checks whether a deadline is temporal and, in that case, returns the initialisation for the `?tick` variable.

$$isTemp(ddl_i) = \begin{cases} [\texttt{?tick/t}] & \text{if } ddl_i \text{ contains one and only one statement} \\ & \quad \text{of the type } \texttt{TEMPORAL(t)} \\ \emptyset & \text{otherwise.} \end{cases}$$

We then modify the $\texttt{tick(}C, m\texttt{)}$ operator so that, for each instance $[id_j, \theta_k]$, it will decrease all the values t such that $[\texttt{?tick}/t] \in \theta_k$ by a value equal to the minimum of t and m. The semantics of $eval(KB, \texttt{TEMPORAL(n)}, \theta_j)$ is then changed to return `true` if and only if the `?tick` variable reaches value zero:

$$eval(KB, \texttt{TEMPORAL(n)}, \theta_j) = \texttt{true} \text{ iff } [\texttt{?tick}/0] \in \theta_j.$$

In other words, for every instance of a norm with a temporal deadline, we activate a timer that is decremented by a call to the function `tick`. The deadline is considered expired when the timer reaches 0. Another consequence of removing the explicit reference to the current time is that, without a reference to the activation time, multiple instances or violations associated with the same pair $(\texttt{nd}i, \theta_j)$ become indistinguishable. This leads to a number of problems at the implementation level. Consider the example in Sect. 5. When the coalition fails to intercept an unauthorized boat `ub` (violation of `nd2`), an instance of `nd3` that binds to `ub` will be activated and included in the list Π. Subsequent violations will bind to the same substitution in the activation condition of `nd3`, preventing any new activation. In order to solve this problem we need to make sure that every new violation of `nd2` will match, for the activation condition of `nd3`, to a substitution that is not currently in Π. We do this by adding a boolean flag in the representation of the violation in the knowledge base. When the first violation of `nd2` associated with θ_j is detected, its description is added to KB with the flag set to `false`. At every subsequent violation associated with the same pair $(\texttt{nd2}, \theta_j)$ we change the value of the flag. We update the semantics of *match* for the construct $\texttt{VIOLATION-OF(}t_1, s_1\texttt{)}$ to include the variable `?flag` bound to the flag value instead of the variable `?violTime`. When, for example, the flag values goes from `false` to `true`, the previous match for the activation of `nd3` is deleted while the instance with `?flag` set to `true` gets activated. This mechanism guarantees that we can activate at least one CTD instance per step for each pair $(\texttt{nd3}, \theta_j)$. Further, to correctly interpret the `VIOLATED` expression, we need to check for a violation of the current instance. Again, without relying on the activation time, we are not able to distinguish between different violations associated to the same pair $(\texttt{nd}i, \theta_j)$. We solve this by adding to the substitution θ_j of each instance $[id_i, \theta_j]$ a variable `?violated` which is initially unbound. We modify Rule R4 (and the equivalent for violated prohibitions) to set `?violated` to `true` when a violation is detected, and update the semantics of *eval* for `VIOLATED` as follows:

$$eval(KB, \texttt{VIOLATED}, \theta_j) = \texttt{true} \text{ iff } [\texttt{?violated/true}] \in \theta_j \tag{5}$$

As a result of these modifications, Rule R4 becomes as follows:

$$\frac{\begin{array}{c} mod_i = O,\ \theta_k = \theta_j \cup [\texttt{?violated/true}] \\ [id_i, \theta_j] \notin \Phi,\ eval(KB, ddl_i, \theta_j) = \texttt{true}, \\ KB^* = addV(KB, [id_i, \theta_j]) \end{array}}{\begin{array}{c} \langle KB, \Delta, [id_i, \theta_j] : I, \Phi, \mathbf{B}, r\rangle \rightarrow \\ \langle KB^*, \Delta, I : [id_i, \theta_k], [id_i, \theta_k] : \Phi, \mathbf{B}, \texttt{true}\rangle \end{array}} \tag{R4*}$$

where θ_k is the substitution obtained by setting the value of the `?violated` flag and *addV* updates the content of KB as discussed above:

$$addV(KB, [id_i, \theta_j]) = \begin{cases} KB \cup \texttt{v(}id_i\texttt{,p(}\theta_j\texttt{),false)} & \begin{array}{l}\text{if } \forall f \in \{\texttt{true}, \texttt{false}\} \\ \quad \texttt{v(}id_i\texttt{,p(}\theta_j\texttt{),}f\texttt{)} \notin KB\end{array} \\ \begin{array}{l} KB \setminus \texttt{v(}id_i\texttt{,p(}\theta_j\texttt{),}f\texttt{)} \\ \quad \cup\, \texttt{v(}id_i\texttt{,p(}\theta_j\texttt{),}\neg f\texttt{)} \end{array} & \text{if } \texttt{v(}id_i\texttt{,p(}\theta_j\texttt{),}f\texttt{)} \in KB \end{cases}$$

6.4 Model Checking Results

We implemented our scenario in Maude and ran the LTL model checker to verify properties of the system for both bounded and unbounded cases.

Table 1 shows the results for bounded model checking[3]. The scenario implemented includes a single UAV a Helicopter and two unauthorized boats and is regulated by norms `nd1`, `nd2` and `nd4`. In all these scenarios agents can perform, according to their capabilities, at most seven actions: start and stop monitoring, start and stop intercepting, start and stop reporting, and move to a different area. We checked the following property, which asks whether a state where `uav` does not monitor the restricted area `area2` always results in a violation of `nd1`:

$$\Box((\neg monitoring(uav1, area2)) \rightarrow violated(1))$$

To prove that this property is always true the model checker has to observe the whole state space, giving us a worst-case scenario in terms of execution time. We can see that both the execution time and the number of states increase exponentially with the number of steps.

Table 2 shows the results for unbounded model checking in different scenarios. **cA** is the number of coalition agents, **uB** the number of unauthorized boats, while for each `nd`i, a ✓ indicates that the norm was included in the scenario.

Table 1. Model checking results: bounded steps

	Step limit				
	7	8	9	10	11
States	4647	12352	32336	81504	202007
Execution time	10 s	29 s	78 s	3 m 8s	8 m

The scenario in row 2 (Table 2.a) is equivalent to that used to produce the results in Table 1. Note that the execution time for bounded model checking at 10 steps is higher than the unbounded case. This is due to the fact that, since we include the time value in KB, conceptually equivalent states are not recognized because their time values differ, making it impossible for the model checker to take advantage of optimizations that rely on state matching.

As we can see from Table 2.a, the scenarios where both `nd2` and `nd3` are enforced are those with higher execution times. We believe this is due to an interaction between temporal deadlines and CTD obligations: In fact `nd3` is a CTD of `nd2` and each of them has a temporal deadline of 3 steps. Values for the `?tick` variable range from 3 to 0 in instances of `nd2` and, whenever `nd2` is violated, the timer for `nd3` is initialized. Our intuition is confirmed by Table 2.b: by decreasing the deadline to 1, we obtain significantly smaller state spaces and execution times.

We now show how model checking can be used to verify that our normative specification is correct, by checking that non compliant behaviours are detected

[3] All tests ran on a Intel Core i5 2.7Ghz, 16 GB RAM.

Table 2. Model checking result: unbounded

Part a: `ddl2 = ddl3 = TEMPORAL (3)`							
cA	**uB**	**nd1**	**nd2**	**nd3**	**nd4**	**States**	**Time**
2	2	✓	✓			5250	20s
2	2	✓	✓		✓	20012	2m
2	2	✓	✓	✓	✓	243994	1h,16m
3	2	✓	✓			19032	2m
3	2	✓	✓		✓	72327	15m
3	2	✓	✓	✓	✓	870165	25h

Part b: `ddl2 = ddl3 = TEMPORAL (1)`							
cA	**uB**	**nd1**	**nd2**	**nd3**	**nd4**	**States**	**Time**
1	2	✓	✓	✓	✓	5717	40s
2	2	✓	✓	✓	✓	17653	5m
3	2	✓	✓	✓	✓	75245	16m

as violations. Let's consider a variation of `nd2` stating that, in order to optimize the allocation of resources, we want one and only one member of the coalition to intercept the unauthorized boat detected in the restricted area. Intuitively we would be tempted to express the norm with the following goal:

$$goal2 \ \texttt{= COUNT (?ag2 IN } \{ \texttt{ memberOf(?ag2,?add)} \\ \texttt{/\textbackslash intercepting(?ag2,?ag1) } \} \texttt{) = 1}$$

which holds true if the number of agents (`?ag2`) that are members of the coalition and are intercepting `?ag1` is equal to 1. We can now use model checking to verify whether this specification captures the meaning we intend. For example, we might ask whether it is true that having two agents intercepting the same boat results in a violation. We refer to `area2` to be the restricted area, `ub` the unauthorized boat, and `uav` and `heli` the UAV and the helicopter respectively. We check the following property, which says that having both `uav` and `heli` intercepting `ub` always results in a violation of `nd2`.

$$\square((\texttt{intercepting(uav,ub)} \wedge \texttt{intercepting(heli,ub)} \\ \wedge \texttt{inArea(ub,area2)}) \rightarrow \texttt{violated(2)})$$

The model checker returns an execution trace that violates the property as a counter example. In fact, if the `uav` and `heli` start intercepting at two different instants of time, the obligation is fulfilled (and thus deleted) when the first agent starts intercepting. We can capture the intended meaning with an obligation to have at least one agent intercepting before the deadline and a prohibition from having multiple agents intercepting the same boat.

We now show, with an example, how model checking can be used to verify robustness-related properties. We want to verify whether compliance with `nd2` and `nd3` guarantees that an unauthorized boat cannot enter and exit the restricted area without being reported or intercepted. We denote by `area1` and `area2` an unrestricted and a restricted area respectively. The following property says that there is no path such that `ub` goes from `area2` to `area1` being neither intercepted nor reported and without triggering a violation of `nd2` or `nd3`.

$$\neg\Diamond(\texttt{inArea(ub,area2)} \wedge \Diamond\texttt{inArea(ub,area1)} \wedge \Box(\neg\texttt{violated(2)} \wedge \neg\texttt{violated(3)} \wedge \neg\texttt{intercepting(uav,ub)} \wedge \neg\texttt{reporting(uav,ub)} \wedge \neg\texttt{intercepting(heli,ub)} \wedge \neg\texttt{reporting(heli,ub)}))$$

The model checker shows as a counterexample a path where `ub` moves from `area2` to `area1` before the deadline for it being intercepted, causing the expiration of `nd2`. We thus verified that our normative system does not guarantee that the specified critical situation will never occur, even if we consider only compliant paths. If we want to make sure that, in a situation of compliance, a boat that exits the area is at least reported, we can modify *exp*2, *ddl*2 and *exp*3 as:

*exp*2 = `VIOLATED` ; *exp*3 = `false`
*ddl*2 = `TEMPORAL(3) \/ NOT EXISTS{inArea(?ag1,?ar)}`

In this way, both the expiration of the temporal deadline or `ub` exiting `area2` before being intercepted trigger a violation of `nd2`, thus activating an instance of `nd3`. By applying model checking we can see that compliance with revised norms `nd2` and `nd3` guarantees that the boat is intercepted or reported.

7 Discussion

The formalism we use to represent norms builds upon a number of approaches to formalise norms for practical applications. For example Tinnemeirer et al. [17] describe the operational semantics of a normative language with support for norms with deadlines and CTD obligations. Hüber et al. [11] adopt an SOS-approach to formalise the norm lifecycle (activation, fulfilment, violation, etc.) and for monitoring the execution of norm-governed systems, which provides the underpinning for a language (NOPL) for programming such systems. Alvarez-Napagao et al. [2] propose a semantics based on production systems for a norm monitoring component that supports norms with deadlines. Similarly, Hindriks and Van Riemsdijk [10] propose a semantics based on *timed transition systems* to keep track of activation, fulfilment and violation of obligation with real time relative deadlines. This semantics could be used for verification purposes, for example with tools such as *Real-Time Maude* [13]. This issue, however, is only discussed briefly by the authors and no details are offered. We complement this existing research by addressing the issue of verifying temporal logic properties of such systems. CÒIR also permits the representation of collective imperatives, which are not considered in existing models defined using semantics at the operational level.

Existing research on the verification of properties of normative systems has focussedon restricted representations of norms, considering only variations of conditional deontic logic, without considering deadlines, event-driven norms, or collective imperatives. Dennis et al. [6], for example, integrate the ORWELL normative language in the MCAPL verification framework in order to verify properties of agents' organisations. In ORWELL norms are represented through *counts as* rules, which label states as compliant or non-compliant by saying that a *brute fact* counts as an *institutional fact* (e.g. a violation) in a certain context. Our results (Table 2), show that, despite using a more expressive representation, verification times are comparable to those reported by Dennis et al. [6].

In research that shares some similarities with ours, Cliffe et al. [5] describe a formalism for specifying obligations with deadlines, permissions and contrary to duty norms. They use answer set programming to verify properties of systems. Their approach is, however, only able to analyse execution traces up to a certain length, and in this regard, is equivalent to bounded model checking.

Ågotnes et al. [1] consider transitions of a Kripke structure that are labelled as compliant or non compliant. It is then possible to use model checking to verify properties of the system under different compliance assumptions. While such a labelling might be expressive enough to represent the kind of norms captured by our formalism, it is not clear how to compute it from a declarative normative specification.

We believe that this mismatch between formalisms used to specify and monitor norms and those used to verify and analyse normative systems makes it difficult to ensure that norms satisfy certain desired properties. Our work attempts to bridge the gap between norm specification, monitoring and verification by providing an executable specification that is verifiable through model checking.

For future research we plan to explore techniques to exploit domain symmetries in order to improve performance and to extend our model to allow agents to issue imperatives at run-time.

8 Conclusion

In this paper we proposed CÒIR, a language for the specification of obligations and prohibitions with support for common features of real world norms, including deadlines, contrary to duty and event-based activation/deactivation. We showed how, thanks to the fact that we allow existential and universal quantification over variables, our formalism can be used to specify common patterns of collective obligations. We then formalized how norms are to be interpreted by means of an operational semantics which we then implemented in Maude. We discussed how the fact that we explicitly represent time in our model leads to an infinite state space, and hence proposed an abstraction that preserves the semantics and makes unbounded model checking decidable. We then used the Maude LTL model checker to validate our normative specification and to verify its robustness to violations.

Acknowledgments. This research was sponsored by Selex ES.

References

1. Ågotnes, T., Van der Hoek, W., Wooldridge, M.: Robust normative systems and a logic of norm compliance. Logic J. IGPL **18**(1), 4–30 (2010)
2. Alvarez-Napagao, S., Aldewereld, H., Vázquez-Salceda, J., Dignum, F.: Normative monitoring: semantics and implementation. In: De Vos, M., Fornara, N., Pitt, J.V., Vouros, G. (eds.) COIN 2010. LNCS, vol. 6541, pp. 321–336. Springer, Heidelberg (2011)
3. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. The MIT Press, Cambridge (1999)
4. Clavel, M., Durán, F., Eker, S., Lincoln, P., et al.: All About Maude - A High-Performance Logical Framework. Springer, Heidelberg (2007)
5. Cliffe, O., De Vos, M., Padget, J.: Modelling normative frameworks using answer set programing. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 548–553. Springer, Heidelberg (2009)
6. Dennis, L., Tinnemeier, N., Meyer, J.-J.: Model checking normative agent organisations. In: Dix, J., Fisher, M., Novák, P. (eds.) CLIMA X. LNCS, vol. 6214, pp. 64–82. Springer, Heidelberg (2010)
7. Dignum, F.P.M., Broersen, J., Dignum, V., Meyer, J.-J.: Meeting the deadline: why, when and how. In: Hinchey, M.G., Rash, J.L., Truszkowski, W.F., Rouff, C.A. (eds.) FAABS 2004. LNCS (LNAI), vol. 3228, pp. 30–40. Springer, Heidelberg (2004)
8. Gasparini, L., Norman, T.J., Kollingbaum, M.J., Chen, L.: Severity-sensitive robustness analysis in normative systems. In: Ghose, A., et al. (eds.) COIN 2014. LNCS, vol. 9372, pp. 72–88. Springer, Heidelberg (2015). doi:10.1007/978-3-319-25420-3_5
9. Grossi, D., Dignum, F.P.M., Royakkers, L.M.M., Meyer, J.-J.C.: Collective obligations and agents: who gets the blame? In: Lomuscio, A., Nute, D. (eds.) DEON 2004. LNCS (LNAI), vol. 3065, pp. 129–145. Springer, Heidelberg (2004)
10. Hindriks, K.V., Van Riemsdijk, M.B.: A real-time semantics for norms with deadlines. In: Proceedings of the 2013 International Conference on Autonomous Agents and Multi-agent Systems, AAMAS 2013, pp. 507–514. International Foundation for Autonomous Agents and Multiagent Systems, Richland (2013)
11. Hübner, J.F., Boissier, O., Bordini, R.H.: A normative organisation programming language for organisation management infrastructures. In: Padget, J., Artikis, A., Vasconcelos, W., Stathis, K., da Silva, V.T., Matson, E., Polleres, A. (eds.) COIN@AAMAS 2009. LNCS, vol. 6069, pp. 114–129. Springer, Heidelberg (2010)
12. Lamport, L.: Real-time model checking is really simple. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 162–175. Springer, Heidelberg (2005)
13. Lepri, D., Ábrahám, E., Ölveczky, P.C.: Timed CTL model checking in real-time maude. In: Durán, F. (ed.) WRLA 2012. LNCS, vol. 7571, pp. 182–200. Springer, Heidelberg (2012)
14. Norman, T.J., Reed, C.: A logic of delegation. Artif. Intell. **174**(1), 51–71 (2010)
15. Plotkin, G.D.: A structural approach to operational semantics. Technical report, DAIMI FN-19, University of Århus (1981)
16. Şensoy, M., Norman, T.J., Vasconcelos, W.W., Sycara, K.: OWL-POLAR: a framework for semantic policy representation and reasoning. Web Semant.: Sci. Serv. Agents World Wide Web **12–13**, 148–160 (2012)

17. Tinnemeier, N., Dastani, M., Meyer, J.J.C., van der Torre, L.: Programming normative artifacts with declarative obligations and prohibitions. In: International Joint Conference on Web Intelligence and Intelligent Agent Technologies, pp. 145–152 (2009)
18. van der Torre, L.: Contextual deontic logic: normative agents, violations and independence. Ann. Math. Artif. Intell. **37**(1–2), 33–63 (2003)
19. Vasconcelos, W.W., Kollingbaum, M.J., Norman, T.J.: Normative conflict resolution in multi-agent systems. Auton. Agents Multi-agent Syst. **19**(2), 124–152 (2009)