Detection of Runtime Normative Conflict based on Execution Scenarios

Mairon Belchior
Computer Science Department
Fluminense Federal University
Niteroi - RJ, Brazil
email: mbelchior@ic.uff.br

Viviane Torres da Silva
IBM Research (on leave from UFF)
Rio de Janeiro - RJ, Brazil
email: vivianet@br.ibm.com

Abstract — **Norms in multi-agent systems are used as a mechanism to regulate the behavior of autonomous and heterogeneous agents and to maintain the social order of the society of agents. Norms describe actions that must be performed, actions that can be performed and actions that cannot be performed by a given entity in a certain situation. One of the challenges in designing and managing systems governed by norms is that they can conflict with another. Two norms are in conflict when the fulfillment of one causes the violation of the other. When that happens, whatever the agent does or refrains from doing will lead to a social constraint being broken. Several researches have proposed mechanisms to detect conflicts between norms. However, there is a kind of normative conflict not investigated yet in the design phase, here called runtime conflicts, that can only be detected if we know information about the runtime execution of the system. This paper presents an approach based on execution scenarios to detect normative conflicts that depends on execution order of runtime events in multi-agent systems. The designer are able to provide examples of execution scenarios and evaluate the conflicts that may arise if those scenarios would be executed in the system. The conflict verification proposed in this paper occurs in the design phase.**

Keywords-Norms; Normative Conflict; Multi-agent Systems; OWL; SWRL.

I. INTRODUCTION

Norms have been used in open multi-agent systems (MAS) as a mechanism to regulate the behavior of autonomous and heterogeneous agents without directly interfering with their autonomy. They are system-level constraints that are independent from the implementation of specific agents and represent the ideals of behavior of these agents [1]. They represent a way for agents to understand their responsibilities and the responsibilities of the others. Norms define what is permitted, prohibited and obligatory. Norm's specification relates entities, the actions that they execute, and the period during while the actions are being regulated [4].

An important issue that must be considered while specifying the norms is the conflicts that may arise between them. Due to the numeral norms that may be necessary to govern a normative MAS, the normative conflict might not be immediately obvious to the system designer. Two norms are in conflict when the fulfillment of one causes the violation of the other, and vice-versa. In other words, the agent will be in a position in which whatever it does (or refrains from doing) will lead to a social constraint being broken [14]. For example, there is a conflict when a norm prohibits an agent from performing a particular action and another that requires the same agent to perform the same action at the same period of time.

There are many approaches in the literature that deal with conflicts between norms in MAS. However, there is another kind of normative conflict not investigated yet in the design phase that can only be detected when we know information about the runtime execution of the system. We will call this kind of conflict as *runtime conflict*. This kind of conflict depends on events that only happen at runtime. For example, let us suppose that *N1* is a norm that prohibits an agent *Ag* from performing the action *Ac* after the execution of action *X*. Moreover, suppose that *N2* is another norm that obligates the same agent to perform the same action before the execution of another action *Y*. The execution of the actions *X* and *Y* are runtime situations and we do not know when they will be performed by the agents in the system. However, analyzing the execution order of them, we can say that if event *Y* would happen first compared to event *X*, we could assert that *N1* and *N2* will be not in conflict. Otherwise, there will be a conflict between *N1* and *N2*. Therefore, if we have information about the time of when the conditions that make the norms active will happen in the system, i.e., information about the moment the events will occur in the systems, it would be possible to detect the existence of the conflict. In this paper, we defined six types of conditions that define the activation period of a norm, which are (i) the execution of an action by an agent, (ii) a fact that become true for an agent, (iii) the fulfillment or (iv) violation of a norm and (v) the activation or (vi) deactivation of a norm. The proposed approach considers norms with before condition, after condition, both of them and norms with no condition.

In this paper, we propose an approach based on execution scenarios to detect normative conflicts that depend on execution order of runtime events in MAS. The system designer may want to evaluate a possible sequence of actions in the system and know if that sequence would cause any normative conflict. The conflict detection approach identifies normative conflicts in case such scenario would be executed in the system. The proposed approach uses Semantic Web technologies, such as, Semantic Web Rule Language (SWRL) rules, OWL DL and SPARQL query language. The Web Ontology Language (OWL) is an expressive knowledge representation language endorsed by the World Wide Web

Copyright (c) IARIA, 2017. ISBN: 978-1-61208-555-5

Consortium (W3C). OWL DL is a sublanguage of OWL that is based on Description Logic (DL), a decidable fragment of the first-order logic [9]. The SWRL is a Horn clause rules extension to OWL [6]. One of the most powerful features of SWRL is its support for built-ins functions to perform operations for comparisons, mathematical, strings, date, and others.

This paper is organized as follows. Section II presents the related work. Section III formalizes the ontology-based norm definition adopted in this paper and, in Section IV, the execution scenario ontology are presented. Section V describes the normative conflict detection approach and gives an example of detection of this kind of conflict. The conclusions and future works are presented in Section VI.

II. RELATED WORK

Several researchers have investigated mechanisms to detect normative conflicts in multi-agent systems. Some of them deals with the identification of direct conflicts, such as in [3][8][12][13]. Direct conflict involves two norms that are associated with the same entity, regulate the same behavior, have contradictory deontic concepts, and are defined in the same context. The detection of this conflict can be done by simply comparing the norm elements (i.e., entity, behavior, context of the norm) in order to check if they apply to the same elements. There are other mechanisms that also detect indirect conflicts, such as in [1][2][7][10][11]. Indirect conflict involves two norms whose elements are not the same but are related. The detection of indirect conflicts can be done only when the relationships among elements of the norms are identified. However, to the best of our knowledge, none of them is able to detect in the design phase normative conflicts that may occur depending on execution order of runtime events.

Lam et al. [7] proposed an approach that uses SWRL and OWL DL to represent norm-governed organizations, including roles and their relationship and various kinds of normative notions, such as permissions, prohibitions, obligations, power and norm violation detection. A conditional norm with deadlines was specified where the condition is only a *xsd:dateTime* associated with either *before* or *after* object properties. However, the authors did not show how to detect a conflict between norms with conditions. Their approach does not allowed a norm to have a relationship with both *before* and *after* properties. Moreover, runtime conditions, such as those described in this paper are not supported. Their approach can only detect direct and indirect conflicts.

Sensoy et al. [11] developed a framework for representing OWL-based policies for distributed agent-based systems called OWL-POLAR. The activation and expiration conditions of a norm in OWL-POLAR are represented by a conjunctive semantic formula, which are facts in the knowledge base. However, the authors did not take into account before and after conditions and their approach cannot detect normative conflicts that may occur in runtime.

Uszok et al. [12] developed a policy framework called KAoS that uses OWL ontology-based representation and reasoning to specify, deconflict, and enforce policies. KAoS supports two main types of norms: (positive and negative) authorization and (positive and negative) obligation. However, KAoS does not provide mechanisms to represent deactivation condition of a norm. Also, before and after conditions are not supported and their framework only detects direct conflicts.

III. NORM DEFINITION

The main classes represented in the norm ontology are *Norm*, *Context*, *DeonticConcept*, *Entity*, *Action*, *Condition*, *FulfillmentStatus* and *ActivationStatus*. The class *Norm* represents a norm definition used as a mechanism to regulate the behavior of agents in MAS and is defined in DL in Figure 1, as follows.

```
Norm ≡ ∀ hasContext.Context ⊓
  =1 hasDeonticConcept.DeonticConcept ⊓
  =1 hasEntity.Entity ⊓
  =1 hasAction.Action ⊓
  ≤1 hasBefore.Condition ⊓
  ≤1 hasAfter.Condition ⊓
  =1 hasActivationStatus.ActivationStatus ⊓
  =1 hasFulfillmentStatus.FulfillmentStatus ⊓
   ∀ hasConflict.Norm
```

Figure 1. Definition of the *Norm* class

According to the definition above, a norm can be related to instances of the classes *Context*, *DeonticConcept*, *Entity*, *Action*, *ActivationStatus* and *FulfillmentStatus* through the object properties *hasContext*, *hasDeonticConcept*, *hasEntity*, *hasAction*, *hasActivationStatus* and *hasFulfillmentStatus*, respectively. It also can be connected to the *Condition* class via two object properties: *hasBefore* and *hasAfter*. Moreover, a norm can have a relationship to order instances of norm by using the *hasConflict* property. The symbol "=" stands for cardinality restriction and "≤" represents the maximum cardinality restriction. The first one specifies the *exact* number of relationships that an individual must participate in for a given property, while the second specifies the *maximum* number of relationships that an individual can participate in for a given property. The symbol "∀" represents universal restriction. It constrains the relationship along a given property to individuals that are members of a specific class. It does not specify the existence of a relationship, i.e., the universal restriction also describes the individuals that do not participate in any relationship.

The class *Context* determines the application area of a norm. Norms can be defined usually in two different contexts: *Environment* and *Organization* contexts. They are defined in the norm ontology as subclasses of the *Context* class, as shown in Figure 2.

```
Organization ⊑ Context
Environment ⊑ Context
```

Figure 2. Subclasses of Context

The class *DeonticConcept* describes behavior restrictions for agents in the form of obligations, permissions and prohibitions. Thus, the individuals *Obligation*, *Permission*

Copyright (c) IARIA, 2017. ISBN: 978-1-61208-555-5

and *Prohibition* were introduced in the norm ontology, and the class *DeonticConcept* was defined as the enumeration of its members using Nominals in DL, as shown in Figure 3.

```
DeonticConcept ≡
  {Obligation,Permission,Prohibition}
```

Figure 3. Definition of the *DeonticConcept* class

The *Entity* class describes the entities whose behavior is being controlled by a norm. An entity is the subject of a norm-controlled action. It has a relationship with a context, via the *actsIn* object property, to determine in which context an entity is acting. The entities represented in this paper are single agents. Instances of the *Entity* class can perform an action in the MAS. Thus, they can have a relationship along the object property *perfomAction* to individuals that are members of the *Action* class. An entity can also participate in a situation, instance of *Situation* class. A situation is one kind of activation condition that represents a fact in the knowledge base (e.g., an agent has a car, lives in New York or is graduated from a college). An agent can participate in zero, one or many situations by using the object property *participateIn*. Activation conditions will be explained latter in this section. The class *Entity* is defined in DL in Figure 4.

```
Entity ⊑ ∀ actsIn.Context ⊓
  ∀ perfomAction.Action ⊓
  ∀ participateIn.Situation
```

Figure 4. Definition of the *Entity* class

The behavior been controlled by the norm is defined by the *Action* class. An action can be performed by individuals that are members of the *Entity* class via *isPerformedBy* object property, which is the inverse property of the *perfomAction* property. The *Action* class is defined in Figure 5.

```
Action ⊑ ∀ isPerformedBy.Entity
```

Figure 5. Definition of the *Action* class

The class *Condition* determines the period during which a norm is active. A norm has a relationship with a condition via two object properties, namely, *hasBefore* and *hasAfter*, which are used to delimitate its activation period. For example, let *n1* and *n2* be two norms, and *n1* is defined to be activated *after* norm *n2* has been fulfilled. Thus, the fulfilment of *n2* is the condition of norm *n1* and the activation period of *n1* is whenever norm *n2* is fulfilled until +infinite.

A norm can have no relationship with any condition. When that happens, its activation period is since the beginning of the system's execution until +infinite, i.e., the norm is always active. There are six types of conditions defined in the norm ontology as subclasses of the *Condition* class. They are *ExecutionOfAction*, *ActivationOfNorm*, *DeactivationOfNorm*, *FulfillmentOfNorm*, *ViolationOfNorm* and *Situation*, and are defined in Figure 6.

```
ActivationOfNorm ⊑ Condition ⊓
    =1 hasRelatedNorm.Norm
DeactivationOfNorm ⊑ Condition ⊓
    =1 hasRelatedNorm.Norm
FulfillmentOfNorm ⊑ Condition ⊓
    =1 hasRelatedNorm.Norm
ViolationOfNorm ⊑ Condition ⊓
    =1 hasRelatedNorm.Norm
ExecutionOfAction ⊑ Condition ⊓
    =1 hasRelatedAction.Action ⊓
    =1 hasRelatedEntity.Entity
Situation ⊑ Condition
```

Figure 6. Definition of the six types of norm condition

Individuals that are members of any of the classes *ActivationOfNorm*, *DeactivationOfNorm*, *FulfillmentOfNorm* and *ViolationOfNorm* must specify a norm that is related to the condition through the object property *hasRelatedNorm*. The class *ExecutionOfAction* was defined as subclass of *Condition* that has exactly one relationship to *Action* and *Entity* classes through *hasRelatedAction* and *hasRelatedEntity* object properties, respectively. The *Situation* class is one type of condition that represents a fact in the knowledge base.

The class *ActivationStatus* represents the activation status of a norm and can be either *activated*, *deactivated* or *none*. When a norm is *activated*, it means the norm becomes active and must be somehow fulfilled. Once a norm is activated, it can be *deactivated* at some time and no action is required anymore. The *none* activation status means that the norm has not been neither *activated* nor *deactivated* yet. All instance of *Norm* are started in the system with *none* value for its activation status. This status is useful to let the agents know about the existences of the norms. The individuals *Activated*, *Deactivated* and *None* were introduced in the ontology, and the class *ActivationStatus* was defined as the enumeration of its members, as shown in Figure 7.

```
ActivationStatus ≡
    {Activated,Deactivated,None}
```

Figure 7. Definition of the *ActivationStatus* class

The class *FulfillmentStatus* describes the fulfillment status of a norm, which can be either *fulfilled*, *violated* or *unknown*. The *unknown* fulfillment status means that the norm has not been neither *fulfilled* nor *violated* yet. For example, let us suppose we have an activated obligation norm stating that a given action must be performed, but that action has not been execute yet. Hence, in that case, the fulfillment status is *unknown*. However, if that action is executed, the fulfillment status will become fulfilled. But, if that norm turns into deactivated and the action has not been executed yet, then the fulfillment status would be violated. All norms are started with *unknown* value for its fulfillment status. The individuals *Fulfilled*, *Violated* and *Unknown* were introduced in the ontology, and the class *FulfillmentStatus* was defined as the enumeration of its members, as shown in Figure 8.

```
FulfillmentStatus ≡
    {Fulfilled,Violated,Unknown}
```

Figure 8. Definition of the *FulfillmentStatus* class

A norm can have a relationship to individuals that are members of the *Norm* class by using the object property

hasConflict, which represents a normative conflict between two instances of norms.

In order to classify the norms regarding their compliance, the *FulfilledObligationNorm*, *ViolatedObligationNorm*, *FulfilledProhibitionNorm*, *ViolatedProhibitionNorm* and *ViolatedPermissionNorm* classes were introduced in the norm ontology as subclasses of the *Norm* class (Figure 9).

```
FulfilledObligationNorm ⊑ Norm ⊓
    hasDeonticConcept.(Obligation) ⊓
    hasFulfillmentStatus.(Fulfilled)
ViolatedObligationNorm ⊑ Norm ⊓
    hasDeonticConcept.(Obligation) ⊓
    hasActivationStatus.(Deactivated) ⊓
    ¬ FulfilledObligationNorm
FulfilledProhibitionNorm ⊑ Norm ⊓
   hasDeonticConcept.(Prohibition) ⊓
   hasActivationStatus.(Deactivated) ⊓
    ¬ ViolatedProhibitionNorm
ViolatedProhibitionNorm ⊑ Norm ⊓
    hasDeonticConcept.(Prohibition) ⊓
    hasFulfillmentStatus.(Violated)
ViolatedPermissionNorm ⊑ Norm ⊓
    hasDeonticConcept.(Permission) ⊓
    hasFulfillmentStatus.(Violated)
```

Figure 9. Classes used to classify the norms according to their compliance

Due to the open word assumption of OWL, in order to the classification of the classes *ViolatedObligationNorm*, *FulfilledProhibitionNorm* work properly, it is necessary to explicitly limit the universe of known individuals of the *FulfilledObligationNorm* and *ViolatedProhibitionNorm* classes by setting them equivalent to the enumeration of their members. Suppose that the *FulfilledObligationNorm* class has two individuals, called *oblig1* and *oblig2*. Thus, the following axiom is added to the norm ontology (Figure 10).

```
FulfilledObligationNorm ≡ {oblig1, oblig2}
```

Figure 10. Limiting the universe of known individuals of the *FulfilledObligationNorm* class

IV. EXECUTION SCENARIO ONTOLOGY

The norm ontology allows to represent an agent performing an action, participating in a situation, and a norm being fulfilled, violated, activated and deactivated. However, in order to detect normative conflict that depends on execution order of runtime events in MAS, it is not enough to know only that those events occurred in the system. More them that, it is necessary to know when such events were executed in the system and if they happened before or after another one. In other words, if we know the moment when each condition of the system norms happens in the system, then it will be possible to ensure if such norms are in conflict or not.

Therefore, the execution scenario ontology extends the norm ontology in order to add the notion of *time*. The moment when a condition of a norm happens in the system is captured by the *hasConditionTime* datatype property and is represented by an integer value. The range of this property is an xsd:integer. The following axiom was added to the *Condition* class (Figure 11).

```
Condition ⊑ ∀ hasConditionTime.Integer
```

Figure 11. Datatype property *hasConditionTime* added to Condition class

The execution scenario ontology also introduced the class *Time* to represent the moment when an action is performed by an agent and when a situation becomes true in the system for an agent. The *Time* class is related to *hasTime* datatype property, which range is an xsd:integer. The following axioms were added to the classes *Entity*, *Action* and *Situation*, and the *Time* class is defined as follows (Figure 12).

```
Entity ⊑ ∀ entityTime.Time
Action ⊑ ∀ actionTime.Time
Situation ⊑ ∀ situationTime.Time
Time ⊑ ∀ hasTime.Integer ⊓
    (∀ timeEntity.Entity ⊔
     ∀ timeAction.Action ⊔
     ∀ timeSituation.Situation)
```

Figure 12. Definion of *Time* class

As described in section III, a norm is activated during a period of time, which is determined by the *Condition* class along with *hasBefore* and *hasAfter* object properties. An agent can perform an action at any time in the system. However, in order to an obligation norm to be fulfilled by the agent, the regulated action must be performed only while the norm is active, i.e., during the period of time where the norm is active. It is necessary to know when the activation period of the norm starts and ends. Therefore, the datatype properties *hasStart* and *hasEnd* were included to the execution scenario ontology. Their domain and range are the *ActivationPeriod* class and xsd:integer, respectively. The class *ActivationPeriod* represents the timeline period during which a norm is active. The following axiom was added to the *Norm* class and the *ActivationPeriod* class is defined as follows (Figure 13).

```
Norm ⊑ ∀hasActvPrd.ActivationPeriod
ActivationPeriod ⊑
  ∀ hasStart.Integer ⊓
  ∀ hasEnd.Integer
```

Figure 13. Definition of *ActivationPeriod* class

In this paper, we are considering that a norm can have at most one before condition and one after condition. Therefore, a norm can have one of the five types of activation intervals showed in Figure 14. The first type is when a norm has no condition and is always active, i.e., its activation interval starts at time zero and lasts until +infinite. The second type refers to a norm associated with only one before condition. This interval starts from zero and lasts until whenever that condition happens in the system. The third type represents a norm with only one after condition and the interval starts whenever that condition happens and lasts until +infinite. The fourth and fifth types refer to a norm associated with both before and after conditions. They differ each other by the moment the conditions happen in the system. If the before

Copyright (c) IARIA, 2017. ISBN: 978-1-61208-555-5

condition happens first, then the norm activation period is characterized by the fourth interval type. Otherwise, the norm activation period is represented by the fifth interval type.

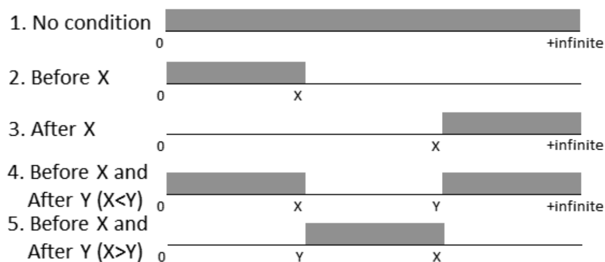

Figure 14. Five types of activation intervals.

The fourth interval type is the only one which a norm has two activation periods, i.e., from zero to whenever the before condition happens and from whenever the after condition happens to +infinite. Therefore, the norms can have one or at most two activation intervals. The classes and properties of the execution scenario ontology are graphically represented in Figure 15.

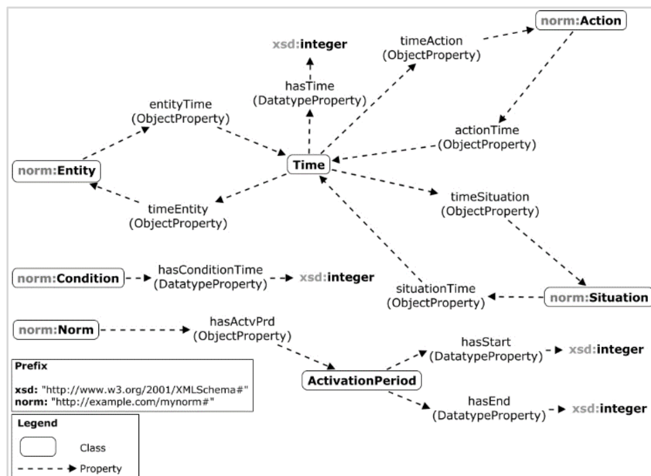

Figure 15. Graphical representation of the execution scenario ontology.

In the norm ontology, six conditions were defined as subclass of *Condition* class, which were the classes *ExecutionOfAction*, *ActivationOfNorm*, *DeactivationOfNorm*, *FulfillmentOfNorm*, *ViolationOfNorm* and *Situation*. The time when such conditions happen in the system can be inferred automatically from the normative system, if the time when an action was performed by an agent and the time when a situation became true to an agent are known in advance. Assuming these times are known, the remaining times can be inferred by using SWRL rules, as follows.

The times of the *ExecutionOfAction* and *Situation* conditions can be easily inferred by using the rules 1 and 2, respectively. These rules verify, respectively, the actions executed and the situations participated by an entity, and check whether the times of those events were provided by the designer. If so, the value of the *hasConditionTime* object property is updated.

```
Rule1: ExecutionOfAction(?c) ∧
hasRelatedAction(?c, ?a) ∧ Action(?a) ∧
hasRelatedEntity(?c, ?e) ∧ Entity(?e) ∧
entityTime(?e, ?t) ∧ Time(?t) ∧
timeAction(?t, ?a) ∧ hasTime(?t, ?ti)
⟶ hasConditionTime(?c, ?ti)
```

```
Rule2: Situation(?c) ∧ participateIn(?e, ?c) ∧
Entity(?e) ∧ entityTime(?e, ?t) ∧ Time(?t) ∧
timeSituation(?t, ?c) ∧ hasTime(?t, ?ti)
⟶ hasConditionTime(?c, ?ti)
```

The norm's fulfillment depends on its deontic concept, i.e., if the norm is an *obligation*, *prohibition* or *permission*. As described in section III, the fulfillment can be *unknown*, *fulfilled* or *violated*. When the norm is an *obligation*, it becomes *fulfilled* when the agent performed the action while the norm is active. If the norm was deactivated, but the agent did not perform the action, then the norm becomes *violated*. If the norm is a *prohibition*, then the opposite behavior can be observed. It becomes *violated* when the agent performed the action while the norm is active, and *fulfilled* when the norm was deactivated, but the agent did not perform the action. When the norm is a *permission*, it becomes *violated* when the agent performed the action, but it has no permission to do that, i.e., the norm is not active. A *permission* norm never becomes *fulfilled* because a permission is an authorization and it is not expected to be perform by the agent.

The condition's time for fulfilment and violation of an obligation norm are shown in rules 3 and 4, respectively. Rule 3 tests whether the *FulfillmentOfNorm* condition is satisfied, i.e., whether an action *?a* was executed by an entity *?e*, both defined by an obligation norm *?n*, and whether *?a* happened during which *?n* was active. In other words, it checks whether *?ti* happened after *?ts* and before *?te* by using the SWRL comparison built-in functions *swrlb:greaterThanOrEqual(?ti, ?ts)* and *swrlb:lessThan(?ti, ?te)*, where *?ti* is the time when action *?a* happened and *?ts* and *?te* are the start and end times of the norm activation period, respectively. As a result, the value of the *hasConditionTime* object property is updated to *?ti*.

```
Rule3: FulfillmentOfNorm(?c) ∧
hasRelatedNorm(?c, ?n) ∧ Norm(?n) ∧
hasDeonticConcept(?n, Obligation) ∧
hasAction(?n, ?a) ∧ Action(?a) ∧
hasEntity(?n, ?e) ∧ Entity(?e) ∧
entityTime(?e, ?t) ∧ Time(?t) ∧
timeAction(?t, ?a) ∧ hasTime(?t, ?ti) ∧
hasActvPrd(?n, ?ap) ∧ hasStart(?ap, ?ts) ∧
hasEnd(?ap, ?te) ∧
swrlb:greaterThanOrEqual(?ti, ?ts) ∧
swrlb:lessThan(?ti, ?te)
⟶ hasConditionTime(?c, ?ti) ∧
hasFulfillmentStatus(?n, Fulfilled)
```

Rule 4 tests whether the related norm of a *ViolationOfNorm* condition was classified as *ViolatedObligationNorm*. If so, the value of the *hasConditionTime* object property is updated with the *?te* value, which is the end time of the norm activation period.

Copyright (c) IARIA, 2017. ISBN: 978-1-61208-555-5

Rule4: ViolationOfNorm(?c) ∧ hasRelatedNorm(?c, ?n) ∧ **ViolatedObligationNorm(?n)** ∧ hasActvPrd(?n, ?ap) ∧ **hasEnd(?ap, ?te)** ⟶ hasConditionTime(?c, ?te) ∧ hasFulfillmentStatus(?n, Violated)

In a similar manner, the condition's time for fulfilment and violation of a prohibition norm can be inferred, but in the opposite way. The rules 5 and 6 calculate the condition's time for activation and deactivation of a norm. These rules get the time when a norm starts and ends its activation period, respectively, and update the value of the *hasConditionTime* object property.

Rule5: ActivationOfNorm(?c) ∧ hasRelatedNorm(?c, ?n) ∧ Norm(?n) ∧ hasActvPrd(?n, ?ap) ∧ **hasStart(?ap, ?ts)** ⟶ hasConditionTime(?c, ?ts)

Rule6: DeactivationOfNorm(?c) ∧ hasRelatedNorm(?c, ?n) ∧ Norm(?n) ∧ hasActvPrd(?n, ?ap) ∧ **hasEnd(?ap, ?te)** ⟶ hasConditionTime(?c, ?te)

The start and end times of a norm activation period cannot be inferred by using SWRL, because in SWRL there is no way to check the existence of these relationships: *hasBefore* and *hasAfter*. This verification can be done by using NOT EXISTS filter expression in SPARQL queries [5]. For example, suppose that *n1* is a norm that has only a relationship with *hasBefore* condition, named *c1*. The activation interval of that norm starts at time zero and the end time will depend on whether or not the condition *c1* was satisfied. If not, the end time is unknown. Otherwise, the end time is the time when the condition *c1* became true in the knowledge base.

V. CONFLICT DETECTION

The execution scenario ontology can be used by the system designer as a means for providing an example of execution scenario performed by the agents in the system. The system designer may want to evaluate a possible sequence of actions in the system and know if that sequence would cause any normative conflict. The conflict detection rule uses the times provided by the execution scenario ontology in order to detect conflicts between the norms in case such execution scenario would be executed in the system. As described in section IV, the designer only needs to provide the time when an action would be performed by an agent and when a situation would become true to an agent. The remaining times are automatically calculated.

Two norms are said to be in conflict when they are active, have contradictory deontic concepts (i.e., prohibition *versus* permission or prohibition *versus* obligation), are defined in the same context, govern the same behavior and are executed by the same entity. To detect conflict between two norms that depends on execution order of runtime events, we have to compare the activation periods two by two in order to find intersections between them. Rule 7 shows the detection of normative conflict between an obligation and a prohibition.

Rule7: Norm(?n1) ∧ Norm(?n2) ∧ hasEntity(?n1, ?e) ∧ hasEntity(?n2, ?e) ∧ hasAction(?n1, ?a) ∧ hasAction(?n2, ?a) hasContext(?n1, ?c) ∧ hasContext(?n2, ?c) ∧ **hasDeonticConcept(?n1, Obligation)** ∧ **hasDeonticConcept(?n2, Prohibition)** ∧ hasActvPrd(?n1, ?ac1) ∧ hasStart(?ac1, ?st1) ∧ hasEnd(?ac1, ?ed1) ∧ hasActvPrd(?n2, ?ac2) ∧ hasStart(?ac2, ?st2) ∧ hasEnd(?ac2, ?ed2) ∧ **swrlb:lessThan(?st1, ?st2)** ∧ **swrlb:greaterThanOrEqual(?ed1, ?st2)** ⟶ hasConflict(?n1, ?n2)

This rule verifies if any activation periods of two norms intersects each other by comparing the initial and final times of their activation intervals. If that happen, then they are in conflict. A similar rule must be created in order to identify conflicts between a permission and a prohibition. The consistencies of the norm and execution scenario ontologies and the SWRL rules were checked by making use of existing DL reasoners such as Pellet [15].

A. *Conflicting Norms Example*

This section presents an example of the detection of this kind of conflict. Let us assume a daily home rules for a family with a child called Riley. The following norms are defined for him.

Norm1 (N1): Agent *Riley* is obligated to perform the action *doHomework*.

Norm2 (N2): Agent *Riley* is permitted to perform the action *playGame* after fulfill the norm *Norm1*.

Norm3 (N3): Agent *Riley* is prohibited to perform *playGame* after he performs *haveLunch* action and before the situation *doneLunching* becomes true for him.

Norm4 (N4): Agent *Riley* is obligated to perform the action *cleanRoom* before he performs *haveLunch*.

Norm5 (N5): Agent *Riley* is prohibited to perform the action *playGame* if he violates the norm *norm4*.

Let us suppose now that the designer provided the following execution scenario and wanted to know if there is any normative conflict in case this scenario would be executed in the system.

- *Riley* performs action *doHomework* at time 10;
- *Riley* performs action *haveLunch* at time 20, and;
- The situation *doneLunching* becomes true for *Riley* at time 30.

The activation periods for each norm are depicted in Figure 16. Since *norm1* does not have activation condition, its activation period starts at the beginning of the system's execution and lasts until +infinite, i.e., the norm is always active. The time of the *fulfilment of Norm1* is inferred by *rule 3*, when *Riley* performs the *doHomework* action at time 10. Thus, the activation of *norm2* starts at time 10. According to the execution scenario, *norm3* prohibits *Riley* to perform the action *playGame* from time 20 to 30, and *norm4* obligates him to perform *cleanRoom* action until time 20. The times of the conditions *ExecutionOfAction haveLunch* and *Situation doneLunching* are inferred by rules 1 and 2, respectively.

Since no information about the execution of *cleanRoom* action was mentioned in the execution scenario during which

norm4 was active, the fulfillment status of *norm4* became violated at time 20, when it became deactivated, and it is captured by rule 4. Therefore, the activation of *norm5* starts at this time.

According to the proposed conflict detection approach, *rule7* identifies two normative conflicts, one between the norms *N2* and *N3*, and the other between norms *N2* and *N5*, because they are applied to the same agent and action, have contradictory deontic concept, and their activation intervals intersect each other. Thereby, it provides to the system designer a way to reevaluate the system norms according to an example of an expected execution scenario of the system.

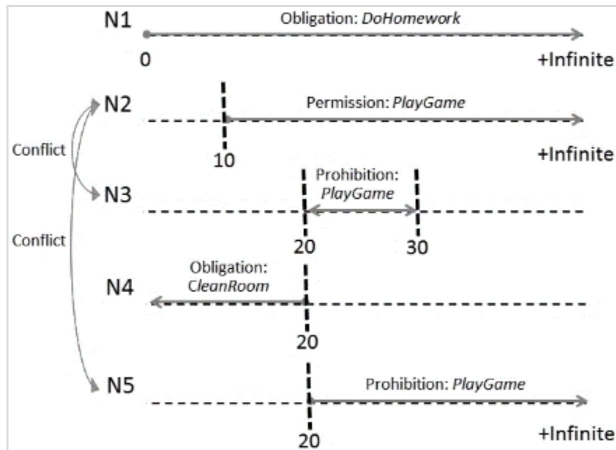

Figure 16. Example of norms in conflict.

VI. CONCLUSIONS

Norms have been used to guide and model the behavior of software agents, without restricting their autonomy. However, conflicts between the norms may occur. A normative conflict arises when the fulfilment of one norm causes the violation of another. In this paper, we have presented an approach based on execution scenarios to deal with the detection of normative conflicts that depends on information about the runtime execution of the system. The designer are able to provide examples of execution scenarios and evaluate the conflicts that may arise if those scenarios would be executed in the system. The conflict verification proposed in this paper occurs in the design phase.

There are several extensions we continue to work on. Since multi-agent systems are composed of multiple autonomous and heterogeneous agents, there is a huge amount of possibilities of execution scenarios to happen in the system. We would like to investigate how the proposed approach can be extended in order to generate automatically the execution scenarios and provide to the designer potential normative conflicts in the system. We also want to extend the proposed approach to support repetition of before and after conditions. Moreover, we are investigating mechanisms for resolution of this kind of normative conflict.

REFERENCES

[1] M. S. Aphale, T. J. Norman, and M. Sensoy, “Goal directed conflict resolution and policy refinement,” In 14th International Workshop on Coordination, Organizations, Institutions and Norms in Agent Systems, Valencia, Spain, 2012, pp. 87-104.

[2] V. T. Da Silva, C. Braga, and J. Zahn, “Indirect Normative Conflict: Conflict that Depends on the Application Domain,” In International Conference on Enterprise Information Systems (ICEIS), 2015, Barcelona, pp. 452-561.

[3] B. F. Dos Santos Neto, V. T. Da Silva, and C. J. P. De Lucena, “Developing goal-oriented normative agents: The NBDI architecture,” In Filipe, J. and Fred, A., editors, Agents and Artificial Intelligence, volume 271 of Communications in Computer and Information Science, pp. 176–191. Springer Berlin Heidelberg, 2013.

[4] K. S. Figueiredo, V. T. Da Silva, and C. O. Braga, “Modeling norms in multi-agent systems with NormML,” In Coordination, Organizations, Institutions, and Norms in Agent Systems VI, volume 6541 of Lecture Notes in Computer Science, pp. 39–57. Springer, Berlin, 2011.

[5] S. Harris, A. Seaborne, and E. Prud’hommeaux, SPARQL 1.1 query language. W3C recommendation, vol. 21, no. 10, 2013.

[6] I. Horrocks et al. SWRL: A semantic web rule language combining OWL and RuleML. W3C Member submission, vol. 21, pp. 79, 2004.

[7] J. S. C. Lam, F. Guerin, W. Vasconcelos, and T. J. Norman, “Representing and Reasoning about Norm-Governed Organisations with Semantic Web Languages,” In Sixth European Workshop on Multi-Agent Systems. Bath, UK. December, pp. 18-32, 2008.

[8] T. Li et al. “Contextualized institutions in virtual organizations,” In Coordination, Organizations, Institutions, and Norms in Agent Systems IX. Springer International Publishing, pp. 136-154, 2014.

[9] S. Rudolph, “Foundations of description logics,” Reasoning Web. Semantic Technologies for the Web of Data. Springer Berlin Heidelberg, 2011, pp. 76-136.

[10] J. S. Santos, and V. T. Silva, “Identifying Indirect Normative Conflicts using the WordNet Database,” In Proceedings of the 18th International Conference on Enterprise Information Systems - Volume 2: ICEIS, ISBN 978-989-758-187-8, 2016 pp. 186-193.

[11] M. Sensoy, T. J. Norman, W. W. Vasconcelos, and K. Sycara, “Owl-polar: A framework for semantic policy representation and reasoning,” Web Semantics: Science, Services and Agents on the World Wide Web, vol. 12, pp. 148–160, 2012.

[12] A. Uszok et al. “New developments in ontology-based policy management: Increasing the practicality and comprehensiveness of KAoS,” In: POLICY ’08: Proceedings of IEEE Workshop on Policies for Distributed Systems and Networks, pp. 145-152, 2008.

[13] W. Vasconcelos, A. García-Camino, D. Gaertner, J. A. Rodríguez-Aguilar, and P. Noriega, “Distributed norm management for multi-agent systems,” Expert Systems with Applications, vol. 39, no. 5, pp. 5990–5999, 2012.

[14] W. W. Vasconcelos, M. J. Kollingbaum, T. J. Norman, “Normative conflict resolution in multi-agent systems,” Autonomous Agents and Multi-Agent Systems, v. 19, n. 2, pp. 124-152, 2009.

[15] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz, “Pellet: A practical owl-dl reasoner,” Web Semantics: science, services and agents on the World Wide Web, vol. 5, no. 2, pp. 51-53, 2007.

Copyright (c) IARIA, 2017. ISBN: 978-1-61208-555-5