Conflict Detection among Multiple Norms in Multi-Agent Systems

Eduardo Augusto Silvestre & Viviane Torres da Silva

To cite this article: Eduardo Augusto Silvestre & Viviane Torres da Silva (2018) Conflict Detection among Multiple Norms in Multi-Agent Systems, Applied Artificial Intelligence, 32:4, 388-418, DOI: 10.1080/08839514.2018.1481591

To link to this article: https://doi.org/10.1080/08839514.2018.1481591

Published online: 04 Jun 2018.

APPLIED ARTIFICIAL INTELLIGENCE
2018, VOL. 32, NO. 4, 388–418
https://doi.org/10.1080/08839514.2018.1481591

Conflict Detection among Multiple Norms in Multi-Agent Systems

Eduardo Augusto Silvestre[a] and Viviane Torres da Silva[b]

[a]Institute of Computing, UFF (Universidade Federal Fluminsene), Niterói, Brazil; [b]IBM Research (on leave from UFF), Rio de Janeiro, Brazil

ABSTRACT
In open multi-agent systems (MAS), norms are being used to regulate the behavior of the autonomous, heterogeneous and independently designed agents. One of the main challenges on developing normative systems is that norms may be in conflict with each other. Norms are in conflict when the fulfillment of one norm violates the other and vice-versa. In previous works, the conflict checkers consider that conflicts can be detected by simply analyzing pairs of norms. However, there may be conflicts that can only be detected when we analyze several norms together. This work presents a conflict checker capable to detect conflicts between two or more norms at the same time. A new, more expressive normative language, represented by a BNF grammar, was developed to define norms and Conflict Checker was implemented in tool format. Two validation principles were applied: software testing and formal verification. The strategy thus developed emerges as a new syntax for definition and verification of conflicts in MAS.

Introduction

Multi-agent systems (MAS) have been gaining great importance in the development of various applications. MAS are autonomous and heterogeneous societies, which can work to achieve common or different goals (Wooldridge 2009).

In order to deal with the autonomy and diversity of interests among different members, the behavior of agents is governed by a set of norms specified to regulate their actions (da Silva 2008). The norms govern the behavior of agents by defining obligations (stating the actions that the agents must perform), prohibitions (stating the actions that the agents must not perform) or permissions (stating the actions that the agents can perform). In a MAS having different goals, an important issue to tackle is that norms can be in conflict with each other. A conflict occurs when norms regulating the same behavior have been activated and are inconsistent (Vasconcelos, Kollingbaum, and Norman 2009).

CONTACT Eduardo Augusto Silvestre eduardosilvestre@iftm.edu.br Institute of Computing, UFF (Universidade Federal Fluminsene), Niterói, Brazil.

Color versions of one or more of the figures in the article can be found online at www.tandfonline.com/uaai.

In such cases, the agent is unable to fulfill all the activated norms. The detection and resolution of conflicts are two of the most challenging area.

Although there are several works that deal with normative conflicts for example (Cholvy and Cuppens 1995), (Elhag, Breuker, and Brouwer 1999), (Kollingbaum et al. 2007), (Vasconcelos, Kollingbaum, and Norman 2009) and (da Silva and Zahn 2013), to the best of our knowledge, all these approaches check for conflicts by analyzing the norms in pairs. However, there are conflicts that can only be detected when we consider several norms together.

For instance, let us consider a conflict that can only be detected if norms N1, N2 and N3 are analyzed together. N1 obliges agent A to dress a red shirt. N2 forbids agent A to dress red pants. N3 obliges agent A to dress pants and a shirt of the same color. There are no conflicts between the pairs N1-N2, N2-N3 and N1-N3, but when the three norms are analyzed together, we can figure out the conflict.

To detect these conflicts it is necessary to generate all possible combinations of norms. This is a NP-complete problem (Shoham and Tennenholtz 1995) (Morales et al. 2014). The first challenge of our approach was to deal with this problem. In order to overcome such an issue, we have defined several filters to divide the norms in subsets. Norms that are not in the same subset are not in conflicts. The conflicts must be checked only among norms of the same subset.

The *second* challenge of the approach was to be able to check for conflicts among norms having potentially different deontic concepts, i.e. be able to check for conflicts among several obligations, permissions and prohibitions. This problem was solved by applying deontic transformations (von Wright 1951) with the goal to normalize the deontic concepts of the norms. All norms were transformed to permissions.

Our *third* challenge was to check for the direct conflicts between the normalized norms of the same subset. The solution for this problem was to find out a paticular situation where all actions being regulated by the permission could be executed. If such a situation existed, there would be no conflict between the norms.

The remainder of this paper is organized as follows. Section 2 presents the background material about the definition of norms. Section 3 describes the multiple norm conflict checker by detailing the approaches to solve our three challenges. Section 4 presents details about the implementation of MuNoCC tool. Section 5 describes the approach used for validation. Section 6 analyzes some of the main works related to this research. Finally, Section 7 states some conclusions and future work.

Norms

Norms have been vastly used in open MAS to cope with the heterogeneity, autonomy and diversity of interests among the different members. Norms describe the behavior that can be performed, that must be performed and that cannot be performed.

Our norm definition is based on (da Silva et al. 2011). In (da Silva et al. 2011), the authors analyze the key strategies found in the literature for describing a norm. According to (da Silva et al. 2011), a norm prohibits, permits or obliges an entity to execute an action in a given context during a certain period of time. The difference between our norm definition and the one presented in (da Silva et al. 2011) is the representation of the action being regulated. In (da Silva et al. 2011), the action is represented by a single constant (e.g. pursue, reach …). Our representation is more expressive. Furthermore, we consider only action for the behavior; we are not considering states. From now, it will only be used the term action.

For our definition of norm, consider the following definitions for sets: Nrm is the set of all norms, C is the set of all contexts, E is the set of all entities, A is the set of all actions, Cd is the set of all activations and deactivations condition, O is the set of all organizations, Env is the set of all environments, Ag is the set of all agents and R is the set of all roles.

A norm $n \in Nrm$ is a tuple of the form:

$$(deoC, c, e, a, ac, dc)$$

where deoC is a deontic concept from the set {O, F, P}, respectively, obliged, forbidden and permitted; $c \in C$ is the context where the norm is defined; $e \in E$ is the entity whose action is being regulated; $a \in A$ is the action being regulated; $ac \in Cd$ indicates the condition that activates the norm and $dc \in Cd$ is the condition that deactivates the norm.

Every norm is defined in the scope of a context. The entity, whose action is being regulated, must fulfill the norm when executing in the context where the norm is being defined. In this paper, we consider that a norm can be defined in the context of an organization $o \in O$ or of an environment $env \in Env$. The set of possible contexts is defined as $C = O \cup Env$. A norm regulates the action of an agent $a \in Ag$, an organization (or group of agents) $o \in O$ or a role $r \in R$. Agents, organizations and roles are entities of the set $E = Ag \cup R \cup O$.

The activation and deactivation conditions, $ac \in Cd$ and $dc \in Cd$, can state an event that can be a date, the execution of an action, the fulfillment of a norm, etc. In this paper, we will focus on the specification of dates because it is easier to figure out which event has occurred first. Thus, we use simple mathematic symbols for example $\leq$ and $\geq$ to indicate that an event occurs before or after another ($\forall n \in N$, $ac \leq dc$).

An action is defined by the name of the action and, optionally, an object where the action will be executed and a list of attributes (with their values). Thus, in this paper, we define four different ways to represent the action:

(i) action;
(ii) action object;
(iii) action (attribute1 = {value1}, attribute2 = {value2}, ...});
(iv) action object (attribute1 = {value1}, attribute2 = {value2},...}).

The designer of a MAS can set any of the types of norms to represent your domain. These different ways of defining a norm represent a great flexibility in creating a MAS. In order to exemplify these four ways to describe an action, let us consider the following four prohibition norms:

(i) Na forbids agent A to get dressed;
(ii) Nb forbids agent A to dress pants;
(iii) Nc forbids agent A to dress red;
(iv) Nd forbids agent A to dress red pants.

The actions described in the norms are represented as:

(i) Na: dress;
(ii) Nb: dress pants;
(iii) Nc: dress (color = {red});
(iv) Nd: dress pants (color = {red}).

```
<norm>:: = ‘<‘<deontic_concept>‘,’ <action>‘,’ <context>‘,’ <entity>‘,’ <activation_date>‘,’
<deactivation_date>‘>‘
<deontic_concept>:: = ‘OBLIGED’ | ‘FORBIDDEN’ | ‘PERMITTED’
<context>:: = ‘ORGANIZATION’ | ‘ENVIRONMENT’
<entity>:: = ‘AGENT’ | ‘ROLE’ | ‘ORGANIZATION’ | ‘ALL’
<action>:: = < action_name> | <action_name> <information_of_the_action>
<information_of_the_action>:: = < object> | ‘(‘<attributes_and_values>‘)’ | <object>
‘(‘<attributes_and_values>‘)’
<attributes_and_values>:: = < attribute>‘ = {‘<values>‘}’ | <attribute>‘ = {‘<values>‘},’
<attributes_and_values>
<values>:: = < value> | <value>‘,’<values>
<action_name>:: = Identifier
<object>:: = Identifier
<attribute>:: = Identifier
<value>:: = Identifier
<activation_date>:: = < date>
<deactivation_date>:: = < date>
<date>:: = < month> <year>
:: = Numbers ‘/’
<month>:: = Numbers ‘/’
<year>:: = Numbers
Numbers = {Number}+
Identifier = {Letter}+
```

We present our normative language described as a BNF grammar with the aim to formally describe the norm. The grammar is represented in the syntax of GOLD Parser Builder (Builder 2015).

The grammar represents all data that may exist in the norm (according to the presented syntax). The grammar is able to represent the four types of norms defined by the language. All norms start from a definition of the norm (<*norm*>) and from this definition, it is possible to fill the data of the norm. The deontic concept (<*deontic_concept*>), the context (<*context*>) and the entity (<*entity*>) are generated from constants. The activation condition (<*activation_date*>) and deactivation condition (<*deactivation_date*>) are generated from dates. The action (<*action*>) is the part of the norm with greater complexity. It is always formed by a name (<action_name) and can also be composed of other elements like objects (<*object*>), attributes (<*attribute*>) and values (<*value*>). Recursion is used for creating norms of more expressiveness.

Conflict checker

The main goal of this section is to present the approach through which one can detect direct conflicts among multiple norms. This section is divided in five subsections. Section 3.1 focuses (a) on identifying the type of norms, which must be considered when checking for conflicts among multiple norms and the ones, which must only be considered when analyzing the norms in pairs, and (b) on stating the types of norms, which can be compared when checking for conflicts. Section 3.2 focuses on presenting the strategies used while checking for conflicts among multiple norms; in 3.2.1, we present the strategy to separate the norms in sets in order to reduce the complexity of the conflict checker; in 3.2.2, the approach that transforms all norms into permissions and in 3.2.3, we check if the norms “intersect”, its if there is a situation where all actions being regulated by the permission can be executed. Section 3.3 presents an application of the strategy over an example. Section 3.4 presents the main algorithms used on checking for conflicts. Section 3.5 presents some considerations about the complexity of the conflict checker.

Combination of norms

The four different ways to represent the actions that were described in Section 2 generate the four types of norms (from (i) to (iv)) being considered in this section. Not all types can be considered together when checking for conflicts among multiple norms, as follows:

- Norms of TYPE (i) do only describe the actions being regulated (without mentioning attributes and objects); thus, it must be analyzed together with norms of any type. For instance, let us consider Na, where agent A is forbidden to dress. Such a norm will conflict with any other norm that obliges the agent to dress, independently of attributes and objects.

- Norms of TYPE (ii) must be analyzed together with norms of TYPE (ii) and TYPE (iv) (besides TYPE (i)), since norms of these types also specify an object. For instance, let us consider Nb, where an agent A is forbidden to dress pants. Such a norm conflicts with an obligation to dress pants (example of TYPE (ii)) and conflicts with an obligation to dress red pants (example of TYPE (iv)). However, such a norm, Nb, does not conflict with an obligation to dress red clothes (example of TYPE (iii)), because the obligation does not mention the object pants.
- Norms of TYPE (iii) must be analyzed together with norms of TYPEs (iii) and TYPE (iv) (besides TYPE (i)), because norms of these types do also describe attributes. For instance, let us consider Nc, which prohibits agent A to dress red clothes. Such a norm conflicts with an obligation to dress red clothes (example of TYPE (iii)) and conflicts with an obligation to dress red pants (example of TYPE (iv)). However, such a norm, Nc, does not conflict with an obligation to dress pants (example of TYPE (ii)).
- Norms of TYPE (iv) use a complete representation of the norm (an action applied over a object with some attributes and values); thus, it must be analyzed together with norms of any type when checking for conflicts. For instance, let us consider Nd where an agent A is forbidden to dress red pants. Such a norm conflicts with an obligation do dress red pants (example of TYPE (iv)).

From the set of four types of norms, only norms of TYPE (iv) must be considered when checking for direct conflicts among multiple norms because the structure of the actions used by this type of norms makes possible the identification of objects and their attributes. In order to find conflicts among multiple norms, it is necessary to find similarities among the objects of the norms. Norms N1, N2 and N3 (described in Section 1) exemplify this scenario. N3 obliges agent A to dress pants and a shirt of the same color. Notice that the action is applied over two different objects: pants and a shirt. Thus, it is necessary to have two more norms for analyzing the conflict: a norm that states the possible colors of pants and a norm that states the possible colors of shirts. The relationship among the objects identified in the actions is necessary for the existence of a conflict among multiple norms.

Applied strategy

In this section, we will focus on presenting the part of the conflict checker that is dedicated to detect conflicts among multiple norms, because it is the main goal of this paper. Therefore, the norms being considered in this section are norms of TYPE (iv). Although the focus is only norms of TYPE (iv), the conflict checker developed (Section 3.4) captures all types of conflicts defined in Section 3.1. The conflict-checker algorithm is divided in the following three steps.

Step 1: Applying filters

The first step is responsible for filtering the norms by including them into sets of similar norms. In order to do so, such a step uses three filters. The first filter separates into sets the norms that apply in the same context, the second filter separates into subsets the norms that govern the same entity and the third filter separates into subsets the ones that regulate the same action. Figure 1 illustrates the sets created by applying the three filters.

After applying all filters, only the norms stored in the same set are the ones that may be in conflict. Norms stored in different sets apply in different contexts, govern different entities and regulate different actions; thus, they do not conflict with one another. In Section 3.5, we explain in detail why the complex of the algorithm that checks for conflicts depends on the filtering strategy.

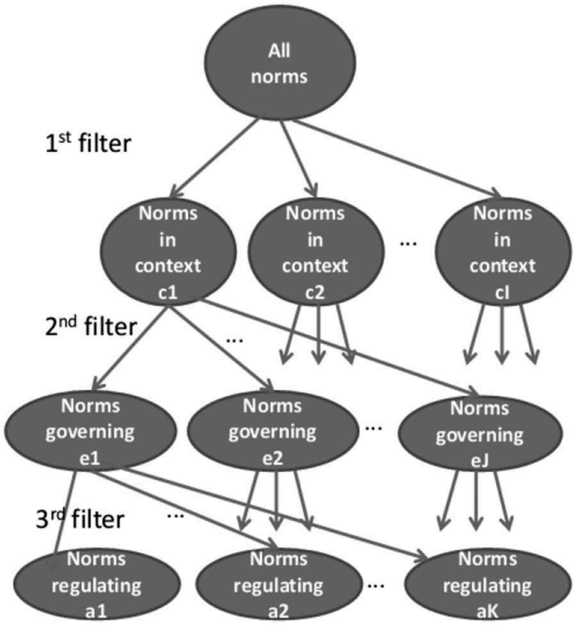


Figure 1. Sets created by the application of the filters.

Step 2: Transforming norms into permissions

The second step of the algorithm is carried out to solve the problem of analyzing several norms with different deontic concepts at the same time. Our strategy to overcome such a problem is to use a single deontic concept to analyze the norms. A copy of the original norms (that are not permissions) with all its deontic concepts transformed to permissions is created. Note that we do not change the original norms, but the copies of such norms.

(1) $Op \leftrightarrow \neg P \neg p$ (2) $Fp \leftrightarrow \neg Pp$

Several approaches studied the deontic transformations. Some approaches use the *O* operator as primitive (McNamara 2014) while others use the *P* operator (von Wright 1951). In this work, we use the *P* operator as primitive and apply the following abbreviations to transform an obligation to a permission (case (1)) and a prohibition to a permission (case (2)):

(a) From obligation to permission

The authors in (von Wright 1951) proposed the weak axiom $Op \rightarrow Pp$ that indicates that when *p* is obliged, *p* is permitted. Following such an axiom and assuming that the designer wants to enable agent *A* to execute *p*, we consider that if there is a norm obliging an agent *A* to execute an action *p*, such a norm can be used as a permission to execute action *p*. Thus, in this step of the algorithm, all obligations are used as permissions.

(b) From prohibition to permission

In this paper, we are using the Closure Principle which says that what is not explicitly forbidden is permitted (Trypuz 2013) (Czelakowski 2015). Therefore, if there is not a prohibition addressed to an agent to execute an action over an object, the agent is permitted to execute such an action over the object.

Following this principal, we consider that if there is a norm prohibiting an agent *A* to execute an action *p*, such a norm states that *A* is not permitted to execute *p*. We assume that it is not necessary to create permissions related to everything that is not said in the prohibition. Thus, in this step of the algorithm, all prohibitions are used as negations of permissions.

Step 3: Checking if norms intersect

The checking for conflicts is executed in the third step of the algorithm. The algorithm checks if the norms in each set are in conflict. Since all norms are permissions, the analysis made by the conflict checker is very simple; it checks if the norms "intersect". Two or more norms intersect if there is at least one possible situation where all the permissions are activated and can be fulfilled. In such a case, the norms are not in conflict because there is a situation where the agent is able to fulfill all the original norms.

The conflict checker starts by checking the norms in a set by pairs of norms and then considers all possible sets of *k*-norms until *k* be equal to the number of norms in the set. At the end, the algorithm has checked for conflicts among all the norms of the set at the same time.

Running example

In order to exemplify our approach, let us consider the three norms described in Section 1. We have augmented these norms by including the context where the norms are executed and the periods during while the norms are activated.

N1: Obliges agent A in orgO to dress a red shirt in 03/01/2015.

N2: Forbids agent A in orgO to dress red pants from 01/01/2015 until 12/31/2015.

N3: Obliges agent A in orgO to dress pants and a shirt of the same color after 02/01/2015.

Norm N3 applies to two objects. It is natural to imagine (without loss of information) that this norm can be divided into two norms: one on pants and one on the shirt (strategy to facilitate visualization and checking for conflicts). As the colors of the pants and the shirt are indifferent, but must be the same color, a variable is used (for convenience, *X*) and this variable must have the same value for pants and the shirt. We get:

N1: (O, orgO, agentA, dress a shirt (color = {red}), 03/01/2015, 03/01/2015)
N2: (F, orgO, agentA, dress pants (color = {red}), 01/01/2015, 12/31/2015)
N3a: (O, orgO, agentA, dress pants (color = {X}), 02/01/2015, _)
N3b: (O, orgO, agent A, dress a shirt (color = {X}), 02/01/2015, _)

In the first step, the algorithm groups all norms in the same set because they are applied in the same context (orgO), govern action of the same entity (agentA) and regulate the same action (to dress). Figure 1 illustrates the set that includes all these norms, where c1 = orgO, e1 = agent A and a1 = to dress.

In the second step, the algorithm transforms the norms to permissions. Remembering that this transformation is not made in the original norms. Let us take a look to norm N2 that prohibits agent A to dress red pants. N2 does not say anything about agent A to dress shirts (of any color), to dress white or black pants or execute the action of writing papers. In short, the prohibition is just about to dress red pants.

In order to transform a prohibition into a permission, we assume that it is not necessary to create permissions related to everything that is not said in the prohibition (since it is already done by applying the Closure Principle). In addition, any permission that talks about actions and objects that are not the ones refereed in the prohibition are not relevant to the checking of conflicts between the prohibition and any other norm. Therefore, a prohibition like N2 is transformed to a permission that only talks about the agent, action, object and attributes described in the norm. N2 is transformed to a norm that permits agent A to dress pants NOT red.

Returning to our example, the transformation of norms N1, N3a and N3b into permissions is very simple since they are obligations, as follows:

N1: (P, orgO, agentA, dress a shirt (color = {red}), 03/01/2015, 03/01/2015)
N3a: (P, orgO, agentA, dress pants (color = {X}), 02/01/2015, _)
N3b: (P, orgO, agent A, dress a shirt (color = {X}), 02/01/2015, _)

The transformation of N2, which is a prohibition, to a permission is done by negating the color of the pants, i.e. the color of the pants is changed to its complement.

N2: (P, orgO, agentA, dress pants (color = {NOT red}), 01/01/2015, 12/31/2015)

In the third step, the checking for conflicts is executed. In our example of N1, N2 and N3, it is easy to see that any group of two norms is not in conflict. Therefore, it is possible to find out situations where all norms can be fulfilled. The conflict only takes place when we consider the three norms together. In 03/01/

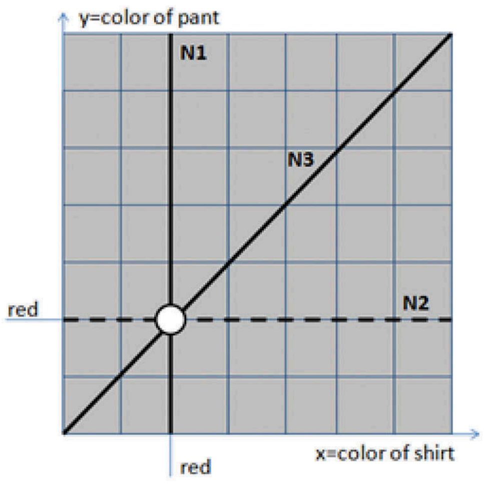


Figure 2. Graph representing the normative conflict.

2015, when the three norms are activated, agentA executing in orgO is permitted to dress a red shirt, a not red pants and a pants and a shirt of the same color.

Figure 2 shows a diagram indicating that there is not an intersection among the norms. The rows represent the colors of pants and the columns represent the colors of shirts. A continuous line (N1 and N3) indicates that there is a permission only in situations, which coincide with the line. A dashed line (N2) indicates that there is not a permission in situations, which coincide with the line. For instance, N1 indicates that there is a permission to dress a red shirt and N3 indicates that there is a permission to dress pants and shirts of same color. Already, N2, a dashed line, indicates that there is not a permission to dress red pants. Figure 2 shows that the three norms do not intersect (represented by a circle). At the only point where norms could intercept, the N2 norm is not permitted.

Algorithms

We now present the main algorithms used. The algorithms were implemented in Java and are available at http://goo.gl/Jh9CV4. The implementation allows creating norms of TYPE (i), TYPE (ii), TYPE (iii) and TYPE (iv), by following the specifications in Section 2. The program is able to detect direct conflicts using any combination of four types of norms. In the following paragraphs, we will present the algorithms that conducted the strategy of the conflict checker.

Algorithm 1 receives a set of norms and calls four other algorithms: to classify the norms in sets (Algorithm 2), convert the norms to permissions (Algorithm 4), check for conflicts among norm with the same type (Algorithm 5) and check for conflicts among norm with the different types (Algorithm 12). The algorithm returns the set of norms in conflict.

```
 Algorithm 1 Function: conflictChecker
Require: norms: list of norms
function conflictChecker(norms)
   lNormsSets ← classifyNormsInSets(norms)
   lPermittedNorms ← convertNormsToPermissions(norms)
   lConflicts ← checkForConflictsEqualTypes(lNormsSets, lPermittedNorms)
   lConflictsDiff ← checkForConflictsDiffTypes(lNormsSets)
   add "lConflictsDiff" to list "lConflicts"
   return lConflicts
endfunction
```

Algorithm 1. Main function of the analysis of conflicts.

Algorithms 2 and 3 classify the norms in sets of related norms. Since all norms in a set are similar, i.e. they apply in the same context, govern the same agent and the same action, each new norm is compared with only one of the norms in the set. If the new norm is similar to the norms in the set, the new norm is included in the set. If the new norm is not similar to the norms in the already created sets, a new set is created to store the new norm.

```
Algorithm 2 Function: classifyNormsInSets
Require: norms: list of norms
function classifyNormsInSets(norms)
   lNormsSets ← new list of list of norms
   for i ← 1 until size(norms) do
      included ← false
      norm ← norms[i]
      if (not lNormsSets.empty) then
         for j ← 1 until size(lNormsSets) do
            tempSet ← lNormsSets[j]
            if (normsAreEquivalent(tempSet[1], norm)) then
                add "norm" to list "tempSet"
                lNormsSets[j] ← tempSet
                included ← true
            endif
         endfor
      endif
      if (not included) then
         tempSet ← new list of norms
         add "norm" to list "tempSet"
         lNormsSets[size(lNormsSets)+1] ← tempSet
      endif
   endfor
   return lNormsSets
endfunction
```

Algorithm 2: Insert, group and classify the norms.

```
Algorithm 3 Function: normsAreEquivalent
Require: norm1: a norm, nom2: a norm
function normsAreEquivalent(norm1, norm2)
   if (not contextChecker(norm1, norm2)) then
      return false
   endif
   if (not entityChecker(norm1, norm2)) then
      return false
   endif
   if (not actionChecker(norm1, norm2)) then
      return false
   endif
   return true
endfunction
```

Algorithm 3: Analysis of the equivalence between the two norms.

Algorithm 4 is responsible to convert the norms to permissions. In order to transform an obligation to a permission, it is only necessary to rewrite its deontic concept to permission. In order to transform a prohibition to a permission, it is necessary not only to rewrite the deontic concept to permission, but also to invert the values of the attributes, as explained in Section 3.2.2.

```
Algorithm 4 Function: convertNormsToPermissions
Require: norms: list of norms
function convertNormsToPermissions(norms)
   for i ← 1 until size(norms) do
      if (norms[i].deonC == "OBLIGATION") then
         norms[i].deonC ← "PERMISSION"
      elseif (norms[i].deonC == "PROHIBITION") then
         norms[i].deonticConcept ← "PERMISSION"
         newAction ← norms[i].action
         newAction ← invertAttributesValues(newAction)
         norms[i].action ← newAction
      endif
   endfor
   return norms
endfunction
```

Algorithm 4: Convert the norms for norms permitted.

Algorithm 5 receives a set of grouped norms and norms converted to permissions. The main responsibility of this algorithm is to send the norms to check conflicts.

```
Algorithm 5 Function: checkForConflictsEqualTypes
Require: lNormsSets: list of norms grouped, lPermittedNorms: list of permitted norms
function checkForConflictsEqualTypes(lNormsSets, lPermittedNorms)
   lConflicts ← new list of list of norms
   for i ← 1 until size(lNormsSets) do
      groupOfNorms ← lNormsSets[i]
      if (size(groupOfNorms) < 2) then
         continue
      endif
      normsRet ← verifyConflictsAmongEqualTypes(normsNtoN, lPermittedNorms)
      add "normsRet" to list "lConflicts"
   endfor
   return lConflicts
endfunction
```

Algorithm 5: Function that checks the conflicts.

Algorithm 6 receives a set of two or more norms at a time and its main functions: classify and distribute. The norms are classified in groups of norms, which have the same type and sent to algorithms.

There is an alogorithm for detecting conflicts for each type of norm, i.e. one for TYPE (i), one for TYPE (ii), one for TYPE (iii) and one for TYPE (iv).

```
Algorithm 6 Function: verifyConflictsAmongEqualTypes
Require: normsNtoN: combination of norms, lPermittedNorms: list of permitted norms
function verifyConflictsAmongEqualTypes(normsNtoN, lPermittedNorms)
   lConflicts ← new list of list of norms
   //creates a structure that classifies the norms by type
   map ← classifyNorms(normsNtoN)
   type1 ← map["TYPE1"] //gets all norms of type 1
   if (size(type1) > 1) then
      normsRet ← verifyConflictsType1(type1)
      add "normsRet" to list "lConflicts"
   endif
   type2 ← map["TYPE2"] //gets all norms of type 2
   if (size(type2) > 1) then
      normsRet ← verifyConflictsType2(type2)
      add "normsRet" to list "lConflicts"
   endif
   type3 ← map.get["TYPE3"] //gets all norms of type 3
   if (size(type3) > 1) then
      normsRet ← verifyConflictsType3(type3)
      add "normsRet" to list "lConflicts"
   endif
   type4 ← map["TYPE4"] //gets all norms of type 4
   if (size(type4) > 1) then
      for i ← 2 until size(type4) do
         normsNtoN ← genAllCombinations(type4, i)
         for j ← 2 until size(normsNtoN) do
            normsRet ← verifyConflictsType4(normsNtoN[j], lPermittedNorms)
            add "normsRet" to list "lConflicts"
         endfor
      endfor
   endif
   return lConflicts
endfunction
```

Algorithm 6: Function that verifies conflicts among norms of same type.

Algorithm 7 is called from the next algorithms to verify if the norms can be fulfilled simultaneously, i.e. if they intersect. The algorithm receives a set of norms and checks if there is an intersection among all periods of activation and deactivation. This algorithm works as a new filter because the process only proceeds to the next step if there is a period where all norms are activated at the same time. The further implementation of this filter takes into account their higher cost (necessary to compare periods of activation and deactivation), the activation period is analyzed for each combination of norms generated and this filter does not apply to group of norms itself, but each combination generated from the norms of the group.

```
Algorithm 7 Function: normsIntersect
Require: norms: a list of norms
function normsIntersect(norms)
   for o ← 1 until size(norms) do
      d1Begin ← norms[o].activationDate
      d1End ← norms[o].deactivationDate
      for i ← 1 until size(norms) do
         d2Begin ← norms[i].activationDate
         d2End ← norms[i].deactivationDate
         ret ← d1Begin <= d2End && d2Begin <= d1End;
         if (not ret) then
            return false
         endif
      endfor
   endfor
   return true
endfunction
```

Algorithm 7: Function that checks intersection among the norms.

Algorithm 8 receives the norms of TYPE (i) of a given group; it generates all the combinations of two norms of entry and checks the conflicts. As the norms have the same context, entity, action and intersect, conflict only exists if the norms have opposite deontic concept (obligation × prohibition and permission × prohibition).

```
Algorithm 8 Function: verifyConflictsType1
Require: norms: list of norms of type 1
function verifyConflictsType1(norms)
   lConflicts ← new list of list of norms
   normsNtoN ← genAllCombinations(norms, 2)
   for i ← 1 until size(normsNtoN) do
      norms ← normsNtoN[i]
      if (not normsIntersect(norms)) then
         continue
      endif
      norm1 ← norms[1]
      norm2 ← norms[2]
      if (deonticConceptChecker(norm1, norm2)) then
         add "norms" to list "lConflicts"
      endif
   endfor
   return lConflicts
```

Algorithm 8: Finction that verifies conflicts among norms of TYPE (i).

Algorithm 9 receives the norms of TYPE (ii) of a given group; it generates all the combinations of two norms of entry and checks the conflicts. As the norms have the same context, entity, action and intersect, conflict exists if the norms have opposite deontic concept (obligation × prohibition and permission × prohibition) and the same object.

```
Algorithm 9 Function: verifyConflictsType2
Require: norm1: a norm, nom2: a norm
function verifyConflictsType2(norm1, norm2)
   lConflicts ← new list of list of norms
   normsNtoN ← genAllCombinations(norms, 2)
   for i ← 1 until size(normsNtoN) do
      norms ← normsNtoN[i]
      if (not normsIntersect(norms)) then
         continue
      endif
      norm1 ← norms[1]
      norm2 ← norms[2]
      if (deonticConceptChecker(norm1, norm2)) then
         if (norm1.objectName == norm2.objectName) then
            add "norms" to list "lConflicts"
         endif
      endif
   endfor
   return lConflicts
endfunction
```

Algorithm 9: Function that verifies conflicts among norms of TYPE (ii).

Algorithm 10 receives the norms of TYPE (iii) of a given group; it generates all the combinations of two norms of entry and checks the conflicts. As the norms have the same context, entity, action and intersect, conflict exists if the norms have opposite deontic concept (obligation × prohibition and permission × prohibition) and exists the same value of attribute in two norms.

```
Algorithm 10 Function: verifyConflictsType3
Require: norm1: a norm, nom2: a norm
function verifyConflictsType3(norms)
   lConflicts ← new list of list of norms
   normsNtoN ← genAllCombinations(norms, 2)
   for i ← 1 until size(normsNtoN) do
      norms ← normsNtoN[i]
      if (not normsIntersect(norms)) then
         continue
      endif
      norm1 ← norms[1]
      norm2 ← norms[2]
      if (deonticConceptChecker(norm1, norm2)) then
         map1 ← norms[1].action.map
         map2 ← norms[2].action.map
         for i ← 1 until size(map1) do
            key1 ← map1.key //an attribute of the norm
            if (not map2.contain(key1))then
               continue
            endif
            valueOfAttr1 ← map1.get(key1)
            valueOfAttr2 ← map2.get(key1)
            interTemp ← valueOfAttr1 ∩ valueOfAttr
            if (interTemp != ∅) then
               add "normsNtoN[i]" to list "lConflicts"
            endif
         endfor
      endif
   endfor
   return lConflicts
endfunction
```

Algorithm 10: Function that verifies conflicts among norms of TYPE (iii).

Algorithm 11 makes the verification of conflicts among multiple norms. It receives the combinations of norms and norms converted to permissions and verify if the norms can be fulfilled simultaneously, i.e. if they intersect. In the algorithm, the mapAttr variable contains the attributes of the norms. For each attribute, we make intersection between the values of such an attribute in all norms of a set. If the intersection is empty, algorithm returns the norms where such an intersection occurred.

```
    Algorithm 11 Function: verifyConflictsType4

    Require: normsNtoN: combination of norms, lPermittedNorms: list of permitted norms
    function verifyConflictsType4(normsNtoN, lPermittedNorms)
       lConflicts ← new list of list of norms
       lConflicts ← preEvaluationOfConflicts(normsNtoN)
       for i ← 1 until size(normsNtoN) do
          norms ← normsNtoN[i] //a combination from a group of norms
          if (not normsIntersect(norms)) then
             continue
          endif
          normsPerm ← new list of norms
          normsPerm ← gets the norms in permitted from lPermittedNorms
//contains the list of all attributes of a norm (e.g., color, …)
          mapAttr ← norms[1].action.map
          inter ← stores the intersections of the norms, the intersection is made for each attribute
          for i ← 1 until size(mapAttr) do
             key ← mapAttr.key //an attribute of the norm
             interTemp ← Ø //an empty set
             for i ← 1 until size(normsPerm) do
//gets the values of an attribute in a specific norm
                valueOfAttr ← normsPerm[i].action.map.get(key)
                interTemp ← interTemp ∩ valueOfAttr
             endfor
//each attribute stores the result of its intersection among the norms
             inter[key] ← interTemp
          endfor
          for i ← 1 until size(mapAttr) do
             key ← mapAttr.key //an attribute of the norm
//if the intersection of an attribute is empty means that exists a conflict
             if (inter[key] == Ø)
                add "norms" to list "lConflicts" //conflicting norms
             endif
          endfor
          return lConflicts
    endfunction
```

Algorithm 11: Function that verifies conflicts among norms of TYPE (iv).

```
    Algorithm 12 Function: checkForConflictsDiffTypes

    Require: lNormsSets: list of norms grouped
    function checkForConflictsDiffTypes(lNormsSets, lPermittedNorms)
       lConflicts ← new list of list of norms
       for i ← 1 until size(lNormsSets) do
          groupOfNorms ← lNormsSets[i]
          if (size(groupOfNorms) < 2) then
             continue
          endif
          if (allNormsHaveSameType(groupOfNorms)) then
             continue
          endif
          normsNtoN ← genAllCombinations(groupOfNorms, 2)
          normsRet ← verifyConflictsAmongDiffTypes(normsNtoN)
          add "normsRet" to list "lConflicts"
          endfor
       endfor
       return lConflicts
    endfunction
```

Algorithm 12 has the same fundamentals of the Algorithm 5, but the focus here is to prepare the norms for verification of conflicts among different types. The group norms will only be redirected if it has at least two norms, and if such norms have an activation period that intersects (Algorithm 7). The algorithm generates all the combinations of two norms for this group; after this step, this group of norms will be sent to conflict detection.

Algorithm 12: Function that checks the conflicts.

Algorithm 13 receives a group of norms, where each element of this group contains two norms and checks if there is conflict between these two norms. The verification of conflicts complies with the description given in Section 3.1. The algorithm checks if the two norms are of different types, and then makes the necessary verification to examine the existence of conflicts.

```
Algorithm 13 Function: verifyConflictsAmongDiffTypes
Require: normsNtoN: combination of norms
function verifyConflictsAmongDiffTypes(normsNtoN)
   lConflicts ← new list of list of norms
   for i ← 1 until size(normsNtoN) do
      if (not normsIntersect(normsNtoN)) then
         continue
      endif
      norm1 ← normsNtoN[1]
      norm2 ← normsNtoN[2]
        //returns the type of the norms
      n1Type ← getBehaviorType(norm1)
      n2Type ← getBehaviorType(norm2)
      //after this, only different types of norms
      if ("NONE" == n1Type or "NONE" == n2Type or n2Type == n1Type) then
         continue
      endif
      if ("TYPE1" == n1Type or "TYPE2" == n2Type) then
         continue
      endif
      if (not deonticConceptChecker(norm1, norm2)) then
         continue
      endif
      if ("TYPE1" == n1Type or "TYPE2" == n2Type) then
         add "normsNtoN[i]" to list "lConflicts"
      elseif ("TYPE2" == n1Type or "TYPE2" == n2Type) then
         if (norm1.actionName == norm2.actionName) then
            if (norm1.objectName == norm2.objectName) then
               add "normsNtoN[i]" to list "lConflicts"
            endif
         endif
      else
         map1 ← norm1.action.map
         map2 ← norm2.action.map
         for i ← 1 until size(map1) do
            key1 ← map1.key //an attribute of the norm
            valueOfAttr1 ← map1.get(key1)
            valueOfAttr2 ← map2.get(key1)
            interTemp ← valueOfAttr1 ∩ valueOfAttr
            if (interTemp != ∅)
               add "normsNtoN[i]" to list "lConflicts"
               break
            endif
         endfor
      endif
   endfor
   return lConflicts
endfunction
```

Algorithm 13: Function that verifies conflicts among norms of different types.

Analysis of algorithms

The computational cost of the conflict checker (Algorithm 1: conflictChecher - the main) is determined by the costs of the calls of the algorithms 2 to 13. The call tree of the algorithms is given by:

```
Computational Cost Conflict Checker
(Begin: Algorithm 1)conflictChecker
   (Begin: Algorithm 2)classifyNormsInSets
      (Algorithm 3)normsAreEquivalent
   (End: Algoritmo 2)classifyNormsInSets
   (Algorithm 4)convertNormsToPermissions
   (Begin: Algorithm 5)checkForConflictsEqualTypes
      (Begin: Algorithm 6)verifyConflictsAmongEqualTypes
         (Begin: Algorithm 8)verifyConflictsType1
            (Algorithm 7)normsIntersect
         (End:Algorithm 8)verifyConflictsType1
         (Begin: Algorithm 9)verifyConflictsType2
            (Algorithm 7)normsIntersect
         (End:Algorithm 9)verifyConflictsType2
         (Begin: Algorithm 10)verifyConflictsType3
            (Algorithm 7)normsIntersect
         (End:Algorithm 10)verifyConflictsType3
         (Begin: Algorithm 11)verifyConflictsType4
            (Algorithm 7)normsIntersect
         (End:Algorithm 11)verifyConflictsType4
      (End: Algorithm 7)verifyConflictsAmongEqualTypes
   (End: Algorithm 5)checkForConflictsEqualTypes
   (Begin: Algorithm 12)checkForConflictsDiffTypes
      (Begin: Algorithm 13)verifyConflictsAmongDiffTypes
            (Algorithm 7)normsIntersect
      (End:Algorithm 13)verifyConflictsAmongDiffTypes
   (End: Algorithm 12)checkForConflictsDiffTypes
(End: Algorithm 1)
```

The computation cost is expressed using the big O notation. Algorithms 3, 4, 6, 8 and 9 are linear (O(n)). Algorithms 2, 7, 10, 11, 12 and 13 are polynomials ($O(n^c)$, where c is constant). Algorithm 5 is the most expensive, because it is exponential ($O(2^k)$). The function *genAllCombinations* makes the algorithm exponential. It was not presented here because it is a well-known combinatorial analysis function. The time and space complexity should be in the order of the number of combinations produced, which is equivalent to the sum of the n-th row of the binomial coefficients in Pascal's triangle or the power set of a set, for example. Thus, the dominant cost of our approach (Algorithm 5) is exponential.

The value of k (complexity of Algorithm 5) comes from the algorithm that groups the norms in sets of similar norms, i.e. norms with the same context, entity and action. The value of k may vary from 1 to n, where n is the number of norms to be checked. Therefore, the best case of the algorithm occurs when $k = 1$. In this case, each set of norms stores exactly one norm, i.e. the norms apply in different contexts and govern different entities and actions. In such a case, the norms are not in conflict and there is not a need to execute the third step of the mechanism.

The worst case of the algorithm occurs when of $k = n$, i.e. all norms are stored in one set. It can happen if all norms apply in the same context, govern the same entity and regulate the same action. The cost of the algorithm in the worst case is $O(2^k) = O(2^n)$, where n is the number of norms. The cost of the medium case is $O(2^x)$, where x is the number of norms in the bigger set. Such an evaluation depends on the application domain. Although it is not possible to calculate the medium case, we strongly believe that the use of filters can drastically reduce the cost of the conflict checker because it is natural to find different contexts, entities and actions in MAS. An SMA is typically defined by a set of agents playing different actions. Therefore, in a SMA it is not reasonable to think that all norms are applied to the same agent and regulate only one action.

MuNoCC (multiple norms conflict checker)

This section provides details of the tool called MuNoCC. The tool was developed using the Java language and the project is available at http://goo.gl/Jh9CV4. The tool provides several types of screens. Some of the most interesting are related the creation of all the components of the norm defined in Section 2. This way, is possible to create visually the norms for conflict detection. Another interesting feature, which will be addressed in depth in the next paragraphs, is the possibility to transform norms inserted in semi-structured language to norms described using definition presented in Section 2.

Figure 3 shows the screen where the user can import norms and check their conflicts. It shows the norms described in Section 3.3 and the conflict that the algorithm found. The authors created a compiler to make the process of translating a norm in semi-structured language to Java objects.

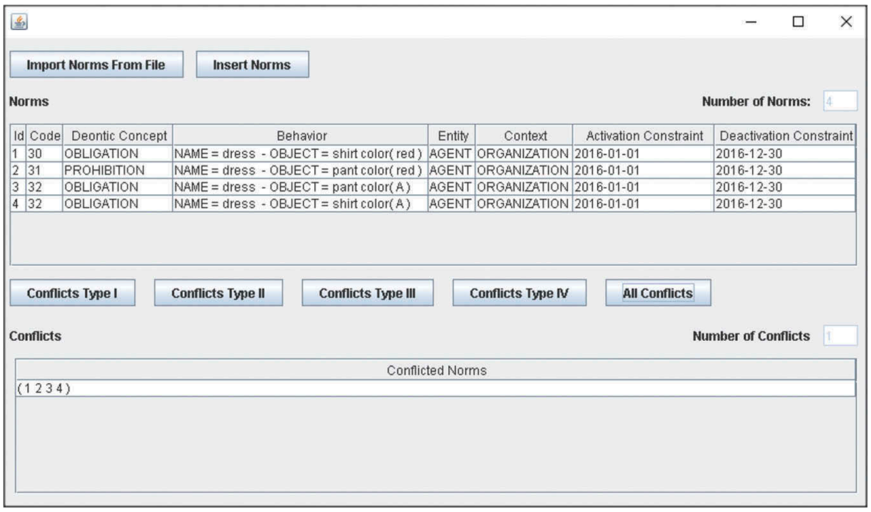
The Lexical analysis is the first phase of the compiler. It takes the user input written in the form of sentences and break the sentences into a series of tokens.

The Syntax analysis is the second phase of the compiler. It takes the input from a lexical analyzer in the form of token streams. The parser analyzes the token stream against the production rules to detect any errors in the code.

The Code generation is the final phase of compilation. It takes the result of syntax analysis and produce java objects instances of class Norm.java. Each line as input produces one instance.

Validation

This section presents the strategies applied for validating the process created and the conflict checker. The main purpose of the validation section is to



Norms

Number of Norms: 4

Id	Code	Deontic Concept	Behavior	Entity	Context	Activation Constraint	Deactivation Constraint
1	30	OBLIGATION	NAME = dress - OBJECT = shirt color(red)	AGENT	ORGANIZATION	2016-01-01	2016-12-30
2	31	PROHIBITION	NAME = dress - OBJECT = pant color(red)	AGENT	ORGANIZATION	2016-01-01	2016-12-30
3	32	OBLIGATION	NAME = dress - OBJECT = pant color(A)	AGENT	ORGANIZATION	2016-01-01	2016-12-30
4	32	OBLIGATION	NAME = dress - OBJECT = shirt color(A)	AGENT	ORGANIZATION	2016-01-01	2016-12-30

Conflicts

Number of Conflicts 1

Conflicted Norms
(1 2 3 4)

Figure 3. Import norms, parse them and detect conflicts.

ensure that: (i) our approach is able to detect all conflicting norms (without false positives and false negatives) (Section 5.1), (ii) the technique is computationally feasible (Section 5.1), (iii) the strategy is applicable to another approaches (Section 5.2) and (iv) the algorithms of the conflict checker are valid such partial and total correctness (Section 5.3).

Case studies

We have designed case studies (detailed in the source code) that execute different tests. In the first case study, which is the simplest one, the authors have manually defined 40 norms (nine norms of TYPE (i), nine norms of TYPE (ii), nine norms of TYPE (iii) and 13 norms of TYPE (iv)) in a domain of clothing. The conflict checker detected 625 conflicts (six conflicts among norms of TYPE (i), six conflicts among norms of TYPE (ii), 10 conflicts among norms of TYPE (iii), 510 conflicts among norms of TYPE (iv), 12 conflicts between norms of TYPE (i) and TYPE (ii), 12 conflicts between norms of TYPE (i) and TYPE (iii), 17 conflicts between norms of TYPE (i) and TYPE (iv), zero conflict between norms of TYPE (ii) and TYPE (iii) (as expected), 17 conflicts between norms of TYPE (ii) and TYPE (iv) and 35 conflicts among norms of TYPE (iii) and TYPE (iv)). The program was able to detect these conflicts without identifying false positives and forgetting false negative conflicts. The data analysis was done manually.

In the second and third case studies, we have developed a tool to randomly generate norms based on the four types (TYPE (i), TYPE (ii), TYPE (iii) and TYPE (iv)). We have defined a set of contexts, entities, actions, activation/ deactivation periods, objects, attributes and values to be used by a random function to generate a predefined number of norms. The function generates the norms by randomly choosing the type of the norm and its elements. This strategy allows the creation of a large set of norms.

In the second case study, detailed in Table 1, we have created several test cases to analyze the program's ability to create norms and detect their conflicts. Each test case created norms of all four types with the same probability function. As detailed in Table 1, we have tested our approach in six test cases. Our intention with the first three test cases, where the random function have generated respectively 50, 100 and 500 norms, was to show that even with 500 norms the algorithm quickly checks for conflicts. It is also possible to notice that the time spent on checking for conflicts is more related to the number of norms in the bigger set than to the number of generated norms.

In the last three test cases, we have called the random function to generate the same number of norms in each test case (1000 norms), but considering different configurations for the elements of the norms. Our intention with these test cases is to demonstrate that the more similar are the norms, the bigger is the time spent to check for conflicts among them. In the fourth test case, the bigger set stored 29 norms; what means that 29 norms have the same context, entity and action. In such a case, it was necessary to make all possible combinations of 29 norms (2^{29}) in order to find out the conflicts that this set may have.

When we increase the number of contexts, entities and actions to be used to generate the 1000 norms (what is the case of the last test case), the number of similar norms decreases. Thus, the number of norms in the bigger set decreases and, consequently, the time spent to check for conflict decreases. These test cases confirm our hypothesis that the filters applied in our approach really helps on improving the performance of the algorithm.

Table 1. Comparative analysis of different test cases.

	Test Cases					
Number elements related to the norm	1	2	3	4	5	6
Norms generated	50	100	500	1000	1000	1000
Contexts	4	4	4	4	4	**8**
Entities	4	4	4	4	4	**8**
Actions	4	4	4	4	4	**8**
Activation periods	4	4	4	4	**8**	4
Deactivation periods	4	4	4	4	**8**	4
Objects	5	5	5	5	5	5
Attributes	3	3	3	3	3	3
Values per attribute	5	5	5	5	5	5
Sets of norms created after the filters	36	50	64	64	64	445
Norms in the bigger set	3	6	16	29	24	7
Conflicts detected	9	27	460	3165	2201	250
Duration (in ms)	183	217	668	**8642**	**1966**	**524**

Table 2. Comparative analysis of test cases in TYPE (iv).

Number elements related to the norm	Test Case 1	Test Case 2
Contexts	2	1
Entities	2	1
Actions	2	1
Activation periods	2	2
Deactivation periods	2	2
Objects	2	2
Attributes	3	3
Values per attribute	5	5
Norms generated	14	14
Sets of norms created after the filters	8	1
Norms in the bigger set	3	14
Conflicts detected	**2**	**9588**
Duration (in ms)	**194**	**4,72,280**

In the third case study, illustrated in Table 2, the function randomly generates only norms of TYPE (iv) in order to focus on the checking of multiple conflicts. Test Case 2 explores the worst scenario, i.e. all 14 norms generated have the same context, entity and action and, therefore, are stored in the same set. In this case, the time spent by the algorithm to check a small set of 14 norms is tremendous. On the other hand, Test Case 1 explores a more feasible scenario where a set of 14 norms is also generated, but now considering different contexts, entities and actions. Since the bigger set only has five norms, the time spent to check for conflicts in Test Case 1 (508 ms) is extremely smaller than the time spent in Test Case 2 (472.280 ms). These test cases also demonstrate the applicability of the filter we have defined.

Rescue-operation system

The main purpose of the validation section is to ensure that the technique can be applicable in a famous scenarios found in literature. The approach was applied in a rescue-operation scenario. The scenario described is adapted from (Vasconcelos, Kollingbaum, and Norman 2009). A simplified non-combatant evacuation scenario in which software agents help humans to coordinate their activities and information sharing. In this scenario, there are two coalition partners, viz., team A and team B, operating within the same area, but each with independent assets. In our scenario, team A has received information that members of a non-governmental organization (NGO) are stranded in a hazardous location. Intelligence has confirmed that these people must be evacuated to a safe location as soon as possible and that the successful completion of this operation takes highest priority.

Team A is based on an aircraft carrier just off the coast and has a number of assets at its disposal, including autonomous unmanned aerial vehicles (AUVs), deployed with sensors to provide on-going visual intelligence for the operation, and helicopters that can be deployed to rescue the NGO workers. Team B is located on land within close distance from the location of the NGO workers. The assets available to team B include ground troops and helicopters.

The most effective plan to complete the rescue mission is to deploy an AUV to provide real-time visual intelligence of the area in which the NGO workers are located, and then to dispatch the helicopter team to uplift the NGO workers and return them to the aircraft carrier. Team A operates under the following norms: (N1) obliged to obtain intelligence using AUV, (N2) permitted to share intelligence and (N3) obliged helicopter to move to the areas 6, 8 and 10. Team B operates under the following norms: (N4) permitted helicopter to move to the areas 5, 7 and 9 and (N5) forbidden helicopter to move to the areas 6, 13 and 15.

Let us suppose that team A discovered that the workers of NGOs are in the area 6 and noticed that its helicopters are not sufficient. They need the help of team B. Since, in the system, there is a norm (N6) that permits helicopters of both teams to move to the same area, team A found that team B could help.

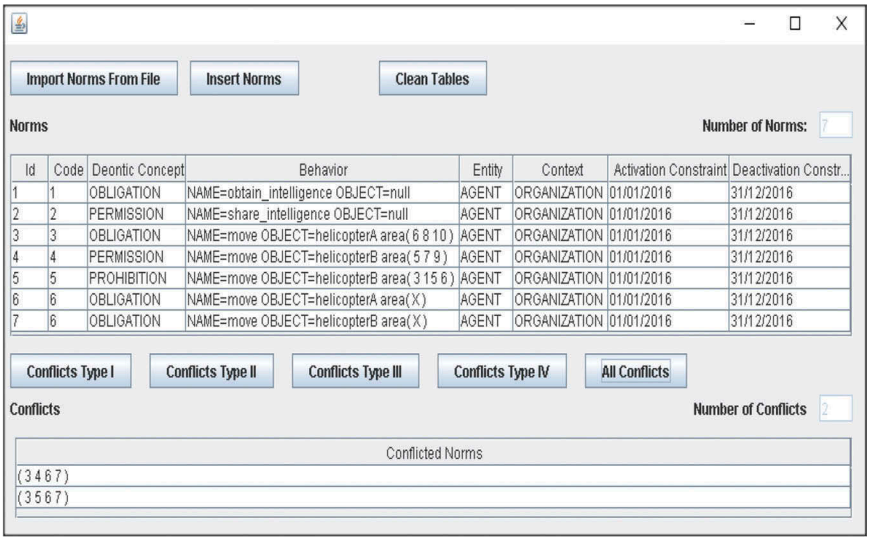
In the defined notation, these norms can be represented as:

N1. O obtain_intelligence_A AUV
N2. P share_intelligence_A AUV
N3. O move helicopter_A area = {6, 8, 10}
N4. P move helicopter_B area = {5, 7, 9}
N5. F move helicopter_B area = {6, 13, 15}
N6. P move helicopter_A e helicopter_B area = X

The application of the conflict checker described in this paper identifies conflicts not found in other approaches in the literature. There is a multiple conflict in norms 3, 5 and 6 and another multiple conflict in norms 3, 4 and 6. In this case, the first parameter is the deontic concept (O, F and P), the second parameter is the action (move), the third parameter is the object (helicopter_A and helicopter_B) and the last parameter is the attribute (area) with its possible values. The constant "X" in norm 6 represents that any value is acceptable, but the value should be the same for helicopter_A and helicopter_B. This example is depicted in Figure 4.

Formal verification

Although the software testing is the primary means to establish the reliability of software, to carry out exhaustive tests is not feasible even for small software. This section presents the application of techniques of formal verification to prove the correctness of the conflict checker. This research uses the Design-by-contract (DBC) (Meyer 1997) technique to prove the correctness of our approach. The DBC paradigm was applied using the concepts of the Java Modeling Language (JML) [18]. The implementation of the JML used



Norms

Number of Norms: 7

Id	Code	Deontic Concept	Behavior	Entity	Context	Activation Constraint	Deactivation Constr...
1	1	OBLIGATION	NAME=obtain_intelligence OBJECT=null	AGENT	ORGANIZATION	01/01/2016	31/12/2016
2	2	PERMISSION	NAME=share_intelligence OBJECT=null	AGENT	ORGANIZATION	01/01/2016	31/12/2016
3	3	OBLIGATION	NAME=move OBJECT=helicopterA area(6 8 10)	AGENT	ORGANIZATION	01/01/2016	31/12/2016
4	4	PERMISSION	NAME=move OBJECT=helicopterB area(5 7 9)	AGENT	ORGANIZATION	01/01/2016	31/12/2016
5	5	PROHIBITION	NAME=move OBJECT=helicopterB area(3 15 6)	AGENT	ORGANIZATION	01/01/2016	31/12/2016
6	6	OBLIGATION	NAME=move OBJECT=helicopterA area(X)	AGENT	ORGANIZATION	01/01/2016	31/12/2016
7	6	OBLIGATION	NAME=move OBJECT=helicopterB area(X)	AGENT	ORGANIZATION	01/01/2016	31/12/2016

Conflicts

Number of Conflicts 2

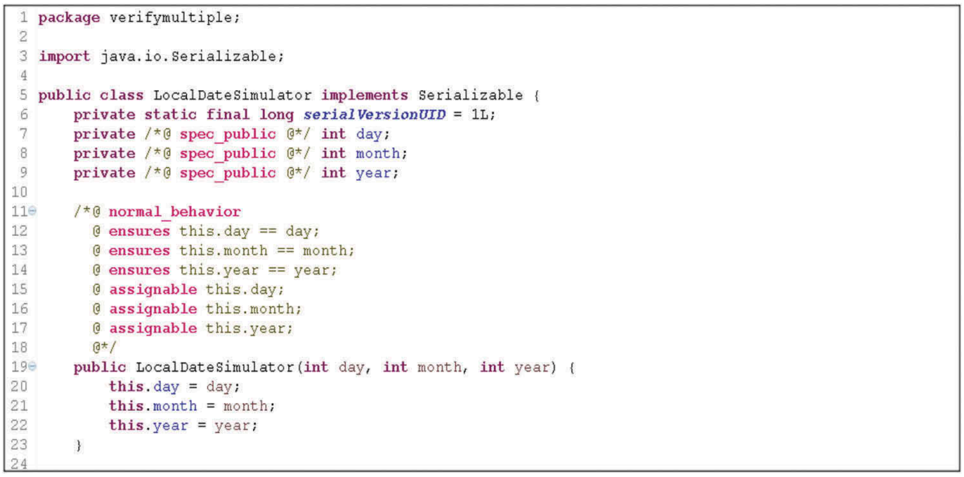
Conflicted Norms
(3 4 6 7)
(3 5 6 7)

Figure 4. Rescue-operation conflicts.

for checking the correctness of the methods was the KeY tool (Ahrendt et al. 2016). It can proof the correctness of a program according to its specification. There are two kinds of proofs: partial correctness (does not require the program to terminate, but when terminates is correct) and total correctness (requires that the program terminate). Details of this implementation are available at http://goo.gl/Jh9CV4.

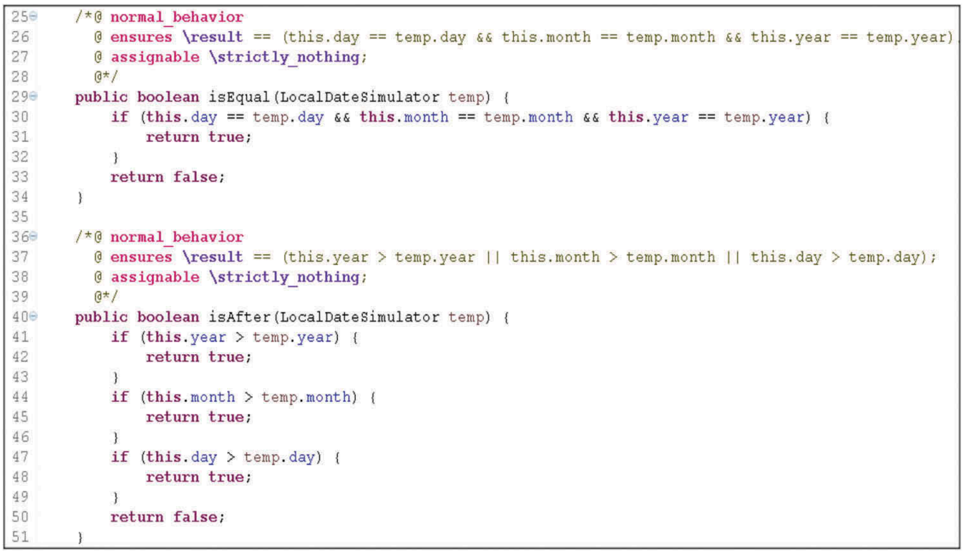
The classes and methods of the conflict checker were annotated with the JML features: preconditions, postcondition, loop invariants and class invariants. After annotating the source code with JML, we conducted the proof of the methods by using KeY.

In order to exemplify our approach, Figures 5, 6 and 7 present an example of implementation that was conducted and checked by the KeY. The figures



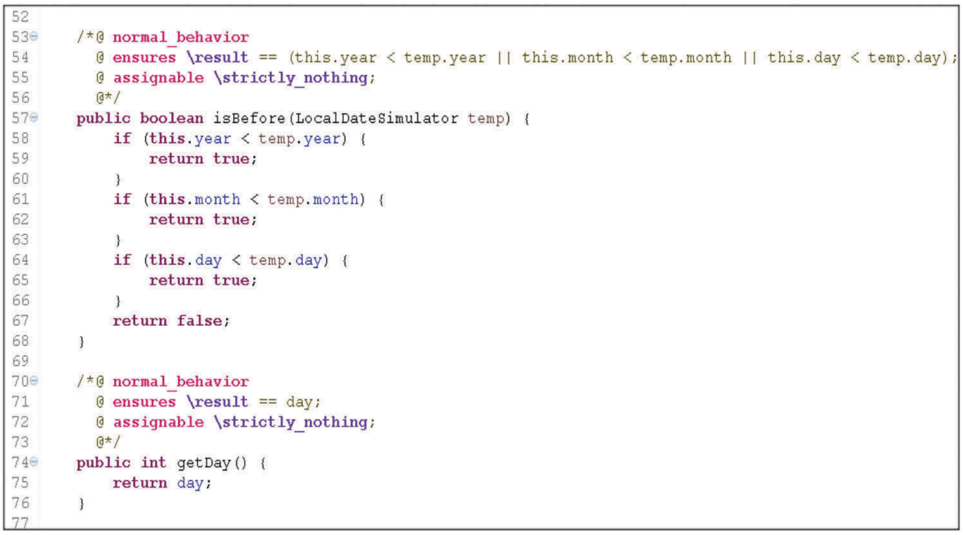
```
package verifymultiple;

import java.io.Serializable;

public class LocalDateSimulator implements Serializable {
    private static final long serialVersionUID = 1L;
    private /*@ spec_public @*/ int day;
    private /*@ spec_public @*/ int month;
    private /*@ spec_public @*/ int year;

    /*@ normal_behavior
      @ ensures this.day == day;
      @ ensures this.month == month;
      @ ensures this.year == year;
      @ assignable this.day;
      @ assignable this.month;
      @ assignable this.year;
      @*/
    public LocalDateSimulator(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
    }

```

Figure 5. Example 1 JML class localdatesimulator.



```java
    /*@ normal_behavior
      @ ensures \result == (this.day == temp.day && this.month == temp.month && this.year == temp.year)
      @ assignable \strictly_nothing;
      @*/
    public boolean isEqual(LocalDateSimulator temp) {
        if (this.day == temp.day && this.month == temp.month && this.year == temp.year) {
            return true;
        }
        return false;
    }

    /*@ normal_behavior
      @ ensures \result == (this.year > temp.year || this.month > temp.month || this.day > temp.day);
      @ assignable \strictly_nothing;
      @*/
    public boolean isAfter(LocalDateSimulator temp) {
        if (this.year > temp.year) {
            return true;
        }
        if (this.month > temp.month) {
            return true;
        }
        if (this.day > temp.day) {
            return true;
        }
        return false;
    }
```

Figure 6. Example 2 JML class localdatesimulator.



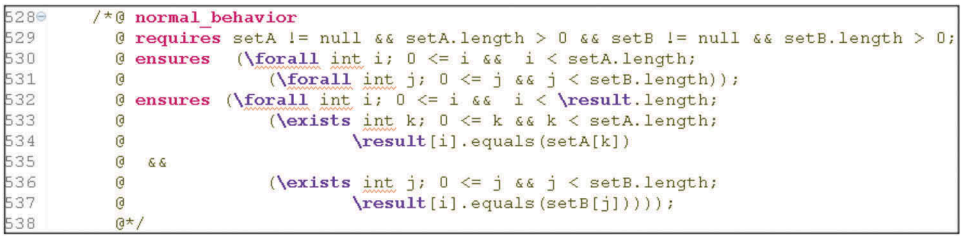
```java

    /*@ normal_behavior
      @ ensures \result == (this.year < temp.year || this.month < temp.month || this.day < temp.day);
      @ assignable \strictly_nothing;
      @*/
    public boolean isBefore(LocalDateSimulator temp) {
        if (this.year < temp.year) {
            return true;
        }
        if (this.month < temp.month) {
            return true;
        }
        if (this.day < temp.day) {
            return true;
        }
        return false;
    }

    /*@ normal_behavior
      @ ensures \result == day;
      @ assignable \strictly_nothing;
      @*/
    public int getDay() {
        return day;
    }
```

Figure 7. Example 3 JML class localdatesimulator.

present a piece of the class LocalDateSimulator that was created to substitute the class LocalDate of Java 1.8. The methods of this class were proved under partial and total correctness.

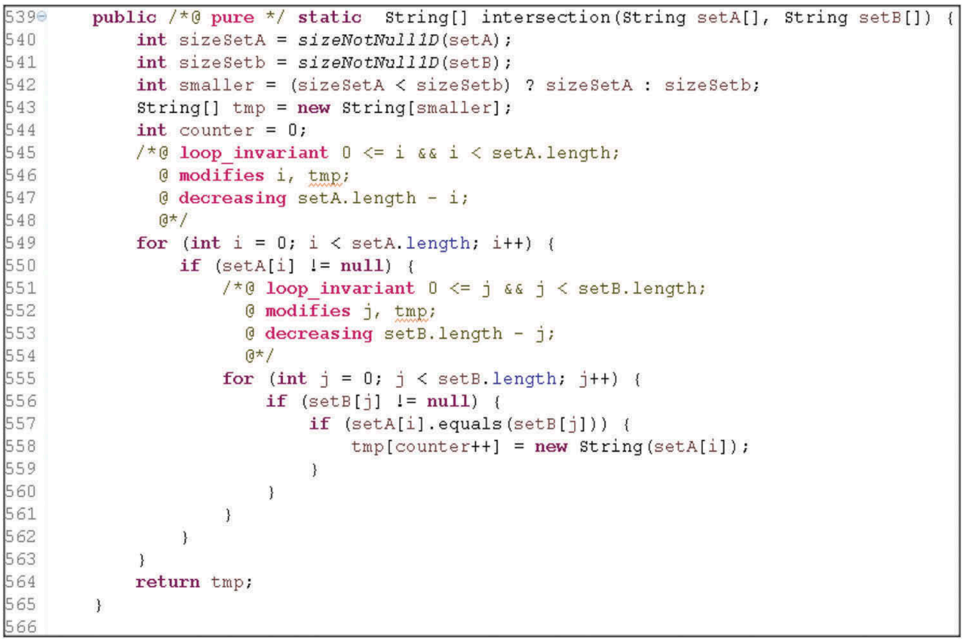
To prove the correctness of algorithms that have loop, one must use the loop invariants in the proof. In JML, a loop specification consists of the following parts: the keyword loop_invariant (specifies a loop invariant), the keyword decreasing (specifies a value which is always positive and strictly decreased in each loop iteration; it is used to prove termination of the loop) and the keyword assignable/modifies (limits the locations that may be changed by the loop).

Figures 8 and 9 show parts of the JML added to Java code for a method. The method takes two vectors of any size and returns a resultant vector of the intersection between the two vectors. Figure 8 shows on line 529 that the vectors must not be null. Lines 530 and 531 ensure that all vectors will be executed to its limits (lower limit and upper limit). From lines 532 to 537, the method ensures that the elements belonging to the resulting vector are both setA and setB vectors. Figure 9 shows the Java code to perform the intersection between vectors. In line 539, the pure keyword indicates that this method does not modify class variables (only local variables). The two JML blocks, shown in lines 544–548 and 551–554, are very similar. The loop invariants used restrict the allowed values for vectors setA and setB. These



```
528      /*@ normal_behavior
529        @ requires setA != null && setA.length > 0 && setB != null && setB.length > 0;
530        @ ensures   (\forall int i; 0 <= i &&  i < setA.length;
531        @                (\forall int j; 0 <= j && j < setB.length));
532        @ ensures  (\forall int i; 0 <= i &&  i < \result.length;
533        @                (\exists int k; 0 <= k && k < setA.length;
534        @                        \result[i].equals(setA[k])
535        @  &&
536        @                (\exists int j; 0 <= j && j < setB.length;
537        @                        \result[i].equals(setB[j]))));
538        @*/
```

Figure 8. Example 1 JML set intersection.



```
539      public /*@ pure */ static  String[] intersection(String setA[], String setB[]) {
540          int sizeSetA = sizeNotNull1D(setA);
541          int sizeSetb = sizeNotNull1D(setB);
542          int smaller = (sizeSetA < sizeSetb) ? sizeSetA : sizeSetb;
543          String[] tmp = new String[smaller];
544          int counter = 0;
545          /*@ loop_invariant 0 <= i && i < setA.length;
546            @ modifies i, tmp;
547            @ decreasing setA.length - i;
548            @*/
549          for (int i = 0; i < setA.length; i++) {
550              if (setA[i] != null) {
551                  /*@ loop_invariant 0 <= j && j < setB.length;
552                    @ modifies j, tmp;
553                    @ decreasing setB.length - j;
554                    @*/
555                  for (int j = 0; j < setB.length; j++) {
556                      if (setB[j] != null) {
557                          if (setA[i].equals(setB[j])) {
558                              tmp[counter++] = new String(setA[i]);
559                          }
560                      }
561                  }
562              }
563          }
564          return tmp;
565      }
566
```

Figure 9. Example 2 JML set intersection.

kinds of invariants are typical in this scenario. The modifies clause indicates the variables that can be modified in the loop scope.

By considering the execution of Java code annotated with JML, KeY was able to prove the partial correctness of the method by ensuring that the list of actions applied to preconditions implies the postconditions. The Key tool was also able to prove the total correctness of the method by ensuring that the variants in lines 547 and 553 decrease in each iteration of the loop and concluding that the algorithm terminates.

One of the most labor-intensive tasks in verifying total correctness of programs is to find the strongest loop invariant for correctness of programs; this task is sometimes undecidable (Blass and Gurevich 2001).

In order to help the authors in such a task, we used the DynaMate tool (Galeotti et al. 2014) to automatically generate loop invariants for the program. However, despite generating several loop invariants for each method, those loop invariants were not different from the set of invariants manually generated by the authors. In (Galeotti et al. 2014), the authors say that from the perspective of software engineering methods, DynaMate can prove are not complex. Unfortunately, the state-of-the-art evidence that proves of large and complex systems is impossible without significant manual effort by highly trained people.

Related work

Several approaches work with normative conflicts in MAS. Some of them deal with the identification of conflicts and others carried out both the identification and resolution of conflicts. However, to the best of our knowledge, all approaches that check for conflicts do only analyze the norms in pairs. There are some papers in the literature that discuss the complexity of considering multiple norms when checking for conflicts, but none of them presents a solution to overcome such a problem.

For instance, the works in (Vasconcelos, Kollingbaum, and Norman 2009) (Cholvy and Cuppens 1995) (Elhag, Breuker, and Brouwer 1999) (Kollingbaum et al. 2007) (da Silva and Zahn 2013) (Kagal and Finin 2007) (Oren et al. 2008) (García-Camino, Noriega, and Juan-Antonio 2007) (Beirlaen, Straßer, and Meheus 2013) focus on the identification of direct and indirect conflicts (i.e. conflicts that occur when the elements of norms being analyzed are not the same, but are somehow related). All of these approaches analyze the norms in pairs when checking for conflicts.

Shoham and Tennenholtz (Shoham and Tennenholtz 1995) evaluate the complexity of norm synthesis, i.e. they evaluate the complexity of finding the set of norms that do not conflict. They attested that the problem is NP-complete through a reduction from 3-SAT. In (Vasconcelos, Kollingbaum, and Norman 2009), the authors also state that the complexity of inserting or removing a norm in a set of norms is exponential in the worst case. Other approaches that evaluate norm synthesis (i.e. (Morales et al. 2014) (Morales et al. 2013) (Christelis and Rovatsos 2009)) are based on heuristics and often intractable in the general case.

These studies have corroborated to show that the detection of conflicts among multiple norms is a very complex activity. Therefore, before thinking in a strategy to find the conflicts, it is necessary to think in a strategy to minimize the complexity of the problem, as we have done in our solution to the problem.

Conclusions and future work

Research in the area of MAS has strongly increased in recent years; in particular, research in normative systems. Despite significant research in the area, there are still many challenges to be considered. In case of normative conflicts, several research problems have not yet been addressed.

After an extensive literature search, several articles were found on the identification of normative conflicts, which is the topic of this paper. Such works focus on the identification of normative conflicts by considering pairs of norms. However, there are conflicts, as the ones exemplified in the text, which can only be detected when checking for conflicts among multiple norms.

The paper presents an approach able to check for conflicts among multiple norms that uses filters and transformations to reduce the computational cost of the algorithm. The main contributions of the paper are:

- An extended specification of norms able to regulate different kinds of actions and their objects (Section 2);
- The specification of filters to divide the norms in sets and make the checking for conflicts quicker (Sections 3.2.1);
- The deontic transformation to deal with norms having the same deontic concept (Section 3.2.2);
- The algorithm to check for conflicts based on the intersection between the (Section 3.4);
- A tool to help MAS designer to check for normative conflicts (Section 4);
- The proof of correctness of the approach (Section 5.3).

A direct consequence of this work is the investigation of how the conflicts among multiple norms should be solved. Can the techniques used to solve conflicts among pairs of norms be used to solve conflicts among multiple norms? An initial approach should investigate the applic-ability of famous techniques found in literature used to solve conflicts among pairs of norms; for example, lex posterior, lex superior and lex specialis; setting priority among norms and association of a value stating the importance of a norm, should also be investigated for the solving of conflicts.

In this work, we have not considered indirect conflicts (da Silva and Zahn 2013). The algorithm presented in this paper does only check for direct conflicts, i.e. conflicts among norms that have the same entities, contexts and actions. An important and necessary extension of our work is the identification of indirect conflict among multiple norms.

References

Ahrendt, W., B. Beckert, R. Bubel, R. Hahnle, V. Klebanov, and P. H. Schmitt. 2016. *The key book: Deductive software verification in practice*. Cham, Switzerland: Springer International Publishing.

Beirlaen, M., C. Straßer, and J. Meheus. 2013. An inconsistency-adaptive deontic logic for normative conflicts. *Journal of Philosophical Logic* 42 (2): 285–315. Springer Netherlands. doi: 10.1007/s10992-011-9221-3.

Blass, A., and Y. Gurevich. 2001. Inadequacy of computable loop invariants. *ACM Transactions Computation Logic* 2 (1): 1–11. New York, NY, USA: ACM. doi: 10.1145/371282.371285.

Builder, G. P. 2015. GOLD Parsing System - Multi-Programming Language, Parser. Accessed 2015 May 15, http://www.goldparser.org/

Cholvy, L., and F. Cuppens. 1995. Solving normative conflicts by merging roles. Proceedings of the 5th International Conference on Artificial Intelligence and Law, 201–09. ICAIL '95. New York, NY, USA: ACM. doi:10.1145/222092.222241.

Christelis, G., and M. Rovatsos. 2009. Automated norm synthesis in an agent-based planning environment. Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 1, 161–68. AAMAS '09. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems. http://dl.acm.org/citation.cfm?id=1558013.1558035.

Czelakowski, J. 2015. *Freedom and enforcement in action: A study in formal action theory*. Dordrecht, Netherlands: Springer Netherlands.

da Silva, F., K. Viviane, T. da Silva, and C. de Oliveira Braga. 2011. Modeling norms in multi-agent systems with normML. Proceedings of the 6th International Conference on Coordination, Organizations, Institutions, and Norms in Agent Systems, 39–57. COIN@AAMAS'10. Berlin, Heidelberg: Springer-Verlag. http://dl.acm.org/citation.cfm?id=2018118.2018122.

da Silva, V. T. 2008. From the Specification to the implementation of norms: An automatic approach to generate rules from norms to govern the behavior of agents. *Autonomous Agents and Multi-Agent Systems* 17(1):113–55. doi:10.1007/s10458-008-9039-8.

da Silva, V. T., and J. Zahn. 2013. Normative conflicts that depend on the domain. *Coordination, Organizations, Institutions, and Norms in Agent* Systems *{IX} - {COIN} 2013 International Workshops, COIN@AAMAS, St. Paul, MN, USA, May 6, 2013, COIN@PRIMA, Dunedin, New Zealand, December 3, 2013, Revised Selected Papers*, 311–26. doi:10.1007/978-3-319-07314-9_17.

Elhag, A. A. O., J. A. Breuker, and B. W. Brouwer. 1999. On the formal analysis of normative conflicts. In *JURIX 1999: The twelfth annual conference*, eds. H. Jaap van den Herik, et al., 35–46. Frontiers in Artificial Intelligence and Applications. Nijmegen: GNI.

Galeotti, J. P., C. A. Furia, E. May, G. Fraser, and A. Zeller. 2014. DynaMate: Dynamically inferring loop invariants for automatic full functional verification. In *Hardware and software: Verification and testing: 10th International haifa verification conference, HVC 2014, Haifa, Israel, November 18-20, 2014. Proceedings*, ed. E. Yahav, 48–53. Cham: Springer International Publishing. doi:10.1007/978-3-319-13338-6_4.

García-Camino, A., P. Noriega, and R.-A. Juan-Antonio. 2007. An algorithm for conflict resolution in regulated compound activities. Proceedings of the 7th International Conference on Engineering Societies in the Agents World VII, 193–208. ESAW'06. Berlin, Heidelberg: Springer-Verlag. http://dl.acm.org/citation.cfm?id=1777725.1777739.

Kagal, L., and T. Finin. 2007. Modeling conversation policies using permissions and obligations. *Autonomous Agents and Multi-Agent Systems* 14 (2): 187–206. Kluwer Academic Publishers. doi: 10.1007/s10458-006-0013-z.

Kollingbaum, M. J., T. J. Norman, A. Preece, and D. Sleeman. 2007. norm conflicts and inconsistencies in virtual organisations. In *Coordination, Organizations, Institutions, and Norms in Agent Systems II*, eds. P. Noriega, J. Vázquez-Salceda, G. Boella, O. Boissier, V. Dignum, N. Fornara, and E. Matson, vol. 4386, 245–58. Lecture Notes in Computer Science. Springer Berlin Heidelberg. doi:10.1007/978-3-540-74459-7_16.

McNamara, P. 2014. Deontic Logic. In *The stanford encyclopedia of philosophy*, ed. E. N. Zalta. Winter, 201. Stanford: Metaphysics Research Lab, Stanford University.

Meyer, B. 1997. Design by contract: Making object-oriented programs that work. {TOOLS} 1997: 25th International Conference on Technology of Object-Oriented Languages and Systems, *24-28 November 1997*, Melbourne, Australia, 360. doi:10.1109/TOOLS.1997.681888.

Morales, J., M. Lopez-Sanchez, J. A. Rodriguez-Aguilar, M. Wooldridge, and W. Vasconcelos. 2013. Automated synthesis of normative systems. Proceedings of the 2013 International Conference on Autonomous Agents and Multi-Agent Systems, 483–90. AAMAS '13. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems. http://dl.acm.org/citation.cfm?id=2484920.2484998.

Morales, J., M. López-Sánchez, J. A. Rodríguez-Aguilar, M. Wooldridge, and W. W. Vasconcelos. 2014. *Minimality and simplicity in the on-line automated synthesis of normative systems*. 109–16. AAMAS '14, Paris, France: IFAAMAS.

Oren, N., M. Luck, S. Miles, and T. J. Norman. 2008. *An argumentation inspired heuristic for resolving normative conflict*. Heidelberg, Germany: Springer-Verlag Berlin Heidelberg.

Shoham, Y., and M. Tennenholtz. 1995. On social laws for artificial agent societies: Off-line design. *Artificial Intelligence* 73(1–2):231–52. doi:10.1016/0004-3702(94)00007-N.

Trypuz, R. 2013. *Krister segerberg on logic of actions*. Dordrecht, Netherlands: Springer Netherlands.

Vasconcelos, W. W., M. J. Kollingbaum, and T. J. Norman. 2009. Normative conflict resolution in multi-agent systems. *Autonomous Agents and Multi-Agent Systems* 19(2):124–52. doi:10.1007/s10458-008-9070-9.

von Wright, G. H. 1951. Deontic Logic. *Mind; a Quarterly Review of Psychology and Philosophy* 60 (237): 1–15. Oxford University Press. doi:10.1093/mind/LX.237.1.

Wooldridge, M. 2009. *An introduction to multiagent systems*, 2nd ed. Hoboken, New Jersey, USA: John Wiley & Sons.