Chapter 2
Specifying and Monitoring Obligations in Open Multiagent Systems Using Semantic Web Technology

Nicoletta Fornara

University of Lugano, via G. Buffi 13, 6900 Lugano, Switzerland
nicoletta.fornara@usi.ch

Abstract. In nowadays open interaction systems where autonomous, heterogeneous and self-interested agents may interact, it is crucial to be able to declaratively specify the norms that regulate the actions of the interacting parties and to be able to monitor their behaviour in order to check whether it is compliant or not with the norms. In this chapter we propose and discuss the advantages of using semantic web languages, tools, and techniques for proposing an application independent model that should be used for the declarative specification and monitoring of obligations. Those obligations are characterized by a class of activation and deactivation events, a class of content actions that may satisfy the obligation and a deadline within which an action belonging to the content class has to be performed. The main contribution of this chapter is to show how it is possible to use semantic web technologies, and in particular OWL 2 DL as formal language for the specification and monitoring of complex obligations and to study how much it is feasible to use an OWL ontology to represent the state of a dynamic open interaction system.

1 Introduction

The specification of *open systems* for the interaction of autonomous agents is widely recognized to be a crucial issue in the development of innovative applications on the Internet, like e-commerce applications, or applications for the management of virtual enterprises. One possible approach to tackle this problem is to model open interaction systems as a set of artificial institutions [2, 1, 20, 11]. Those institutions are devised for the specification of the institutional context where the interaction among autonomous heterogeneous agents may take place. In particular the OCeAN meta-model [12, 9] is mainly composed by: a *communicative part* with the definition of an Agent Communication Language (ACL) whose semantics is defined in terms of social commitments and institutional power [8], a *normative part* for the specification of obligations, prohibitions and permissions [10], and an *organizational part* mainly devoted to the definition of roles.

In this chapter we will mainly focus on the *normative part* and we propose and discuss the advantages of using semantic web languages, tools, and techniques for defining an application independent model for the declarative formal specification and monitoring of *obligations*. In particular we want to be able to specify obligations with the following characteristics. They become active when an event belonging to a specified start event class or to its subclasses happens, this event can be viewed as a condition for obligations activation. A set of possible actions described by means of a more or less detailed class may fulfil those obligations if one of them happens before a given deadline. This is a crucial progress in the flexibility of the normative specification with respect to the solution proposed in [10] where (as better discussed in next section) the content of obligations was a specific action and the time interval for the performance of the action was delimited by fix instant of time. Finally those obligations become cancelled when an event belonging to an end event class happens.

The approach of specifying using a declarative formal language the normative part of a system has many crucial and interesting advantages. In particular it makes possible to represent the norms as *data*, instead of coding them in the software. This has the advantage of making possible to add, remove, or change the norms that regulate the interaction both when the system is off line, and at run-time, without the need to reprogram the interaction system or the interacting agents. Another interesting advantage is that it would be in principle possible to realize agents able to automatically reason on the consequences of their actions and able to interact within different systems without the need of being reprogrammed. Moreover it is possible to realize an application independent *monitoring component* able to keep trace of the state of obligations on the basis of the events that happens in the system and on the basis of agents' actions and capable of reacting to their fulfilment or violation. This is a fundamental component in the architecture of open interaction systems, and may be crucial also in the service oriented architecture [6] and for business process management systems [21]. Another important aspect is that designing a system by using the notion of norm may be very intuitive for human designers and those declarative norms may be more easily understood by human participants of socio-technical systems.

The choice of the formal language used for the declarative specification of normative systems is difficult, crucial, and many aspects have to be taken into account. The most important are: the expressivity of the language, its computational complexity, the fact that the underline logic is decidable, the diffusion of the language among software practitioners and research communities, its feasibility to be used for fast prototyping, and its adoption as an international standard. After many past experiments with other formal languages, in this chapter we decided to adopt OWL (in its OWL 2 DL version[1]), the description logic language recommended by W3C for Semantic Web applications, and more generally semantic web technologies. The main advantage of this choice is that Semantic Web technologies are increasingly becoming a standard for Internet applications and therefore, given

[1] http://www.w3.org/2007/OWL/wiki/OWL_Working_Group

that the OWL logic language is decidable, it is supported by many reasoners (like Fact++, Pellet, Racer Pro, HermiT), tools for ontology editing (like Protégé[2]) and library for automatic ontology management (like OWL-API). Given that it is a standard, it would be easier to achieve a high degree of interoperability of data and applications, which is indeed a crucial precondition for the development of open systems. Finally given that semantic web technologies are becoming very used in innovative applications it will become much easier to teach them to software engineers than convince them to learn and use a logic language adopted by a limited group of researchers.

There are some interesting and challenging problems that may arise from the fact that Semantic Web technologies are not devised for modelling dynamic systems (i.e. systems that changes in time). One is encountered when trying to perform full temporal reasoning; in fact OWL has no temporal operators. Another one is due to the fact that Semantic Web technologies have not been devised to check constrain for example on norm specification, but there are some interesting current studies on how to use the Pellet reasoner for "Simple Integrity Constraints"[3]. A third one is the open-world assumption of OWL logic, it may be a problem for successfully monitoring obligations, that is, when trying to deduce that when the deadline is elapsed an obligation has to be permanently fulfilled or violated.

The added value of this chapter is twofold: the first is to show how it is possible to use semantic web technologies, and in particular OWL 2 DL, as formal language for the specification and monitoring f obligations with activation and deactivation events and deadlines. This model may have many different kinds of applications like the specification of electronic commerce market places, or the monitoring of semantic web services execution, or the flexible specification and monitoring of business process where both software and human agents may interact. The second is to propose to use an OWL ontology not only for the *specification* of a normative systems but also for the *dynamic monitoring* of the state of the interaction among autonomous agents in an open and dynamic environment with respect to a specified set of norms. In particular with this work we are giving our contribution to the open problem of understanding how far the monitoring problem can be solved by using an OWL 2 DL ontology and when it is necessary to integrate it with Java programs.

This chapter is organized as follows. In Section 2 the proposed approach is compared with main alternative approaches. In Section 3 the formal language used in the paper is briefly described. In Section 4 the application independent ontology that can be used to represent and monitor obligations is introduced, discussed and exemplified. In Section 5 some obligations of a concrete case study are formalized using the proposed approach and finally in Section 6 some conclusions are drawn.

[2] See http://www.w3.org/2007/OWL/wiki/Implementations for a complete list of reasoners and tools
[3] http://clarkparsia.com/weblog/category/semweb/owl/pellet/integrity-constraints/

2 Other Approaches

The problem of modelling norms using formal languages is widely recognized as a crucial problem by the multiagent community [3, 19]. Moreover the problem of run-time monitoring those norms is becoming more and more an interesting open question for the multiagent community and for the web service community as demonstrated by various papers on this topic [7, 16, 23, 10]. In particular in [7] Faci et al. propose a framework for non-intrusive monitoring of the state of contract that, similarly to our proposal, is based on the observation of agents' message exchange. Their norms, having a structure quite close to the one proposed in this chapter, are specified using the XML language and their content is specified using ontologies. The main difference between the two approaches is on the monitoring component: in their work it is required to transform the XML representation of norms in another formalism: the augmented transition networks. This transformation presents all the drawbacks that may come from using two different formal languages to specify the same concept in term of consistency, performance, and required knowledge for the engineers who want to adopt this approach. In [16] Lomuscio et al. in order to monitor an agent "all its possible behaviours are represented as a timed automata with discrete data (TADD) and stored in the checker, the monitoring engine checks the snapshots against their TADD specification". One of the main advantages of this approach, as claimed by the authors, is its scalability, this is an important goal to be taken into account and that in our approach can be pursuit by splitting up the state of the interaction in sub-states holding only the information that in a certain moment is relevant for a given interaction. The reference architecture for contract monitoring in e-market scenarios presented in [23] is complementary to the model proposed in this chapter. Finally the main difference between the formalization proposed in this chapter and our previous work on the specification of norms using semantic web technology [10], is that in this chapter the content and the conditions of obligations are specified as classes of actions or events instead as specific action or event.

As discussed in the introduction the choice of using semantic web languages has many advantages and it is a crucial aspect when we compare our work with other ones on norms specifications and properties verification where other formal languages are adopted. Other formal languages are for example the Event Calculus [24, 9], the language for rule specification of the rule engine Jess [13, 4], a variant of Propositional Dynamic Logic (PDL) used to specify and verify liveness and safety properties of multi-agent system programs with norms [5], the Process Compliance Language (PCL) [14].

In literature there are few approaches that use semantic web languages for the specification of norms, even if their importance for the development of flexible security for dynamic and distributed environment is clearly recognized [15]. One interesting approach for policy specification and management is the KAoS framework [18]. In MAS community the word *norm* and *policy* have a similar meaning; a policy could be a positive or negative authorization to perform an action or an obligation. In KAoS, like in the model proposed in this chapter, policies are specified using a set of concepts defined in an OWL DL *core ontology* that could be extended with application dependent ontologies. A crucial difference between the two approaches is the fact that OWL 2 DL is more expressive that OWL DL. Another important difference is in the methods used for monitoring policies: in KAoS policies are usually regimented by means of "guards`` and are monitored by means of platform specific mechanisms.

3 OWL and SWRL

OWL is a practical realization of a Description Logic system known as $\mathcal{SROIQ(D)}$. It allows one to define *classes*, *properties*, and *individuals*. An OWL ontology consists of: a set of class axioms to describe classes, which constitute the *Terminological Box* (*TBox*); a set of property axioms to describe properties, which constitute a *Role Box* (*RBox*); and a collection of assertions to describe individuals, which constitute an *Assertion Box* (*ABox*). Properties can be either *object properties* or *data properties*. Classes can be viewed as formal descriptions of sets of objects (taken from a nonempty universe), and individuals can be viewed as names of objects of the universe. A class is either a *basic class* (i.e., an atomic class name) or a *complex class* build through a number of available *constructors* that express Boolean operations and different types of restrictions on the members of the class.

Through *class axioms* one may specify *subclass* or *equivalence* relationships between classes, that certain classes are *disjoint* (Discla), and that a class is defined by placing restrictions on properties (existential ($\exists$), universal ($\forall$), cardinality, “has-value” ($\ni$), and local reflexivity restrictions. *Property axioms* allow specifying that a given property is the inverse of another property ($^-$), or that a property is functional (Fun), or a transitive property (Tr), or that a property can be obtained by composing properties into *property chains* ($\circ$). Finally, *assertions* allows to specify that an individual belongs to a class, that an individual is related to another individual through an object property, that an individual is related to a data value through a data property, or that two individuals are equal or different.

OWL can be regarded as a decidable fragment of First Order Logic (FOL). The price to pay for decidability, which is considered as an essential preconditions for exploiting reasoning in practical applications, is limited expressiveness. Even in OWL 2 DL (the more expressive version currently under specification) certain useful first-order statements cannot be formalized. Given the limited expressivity of OWL the Semantic Web Rule Language (SWRL)[4] has been proposed to extend the set of OWL axioms to include Horn-like rules of the form of an implication between an antecedent (body) and consequent (head). Recently certain OWL reasoners, like Pellet, have been extended to deal with SWRL rules. To preserve decidability, however, rules have to be used in the *safe mode*, which means that before being exploited in a reasoning process all their variables must be instantiated by pre-existing individuals. An important aspect of SWRL is the possibility

[4] http://www.w3.org/Submission/SWRL/

of including *built-ins*, that is, Boolean functions that perform operations on data values and return a truth value. In what follows we use capital initials for classes and lower case initials for properties and individuals, we assume that all the individuals introduced are different.

4 An Application Independent Ontology for Modelling and Monitoring Agents' Interactions

In this section we introduce the classes, the properties, and the axioms of the *application independent* part of the ontology ("*upper ontology*") that one has to use to specify and monitor agents' obligations in those applications where the realization of an *open normative interaction system* is required. In order to completely formalize a real interaction system, as exemplified in Section 4.2, this ontology has to be extended with application dependent classes, properties, and axioms that are used to model the application dependent actions and events that appear in the content or in the condition of obligations.

In particular we first describe the OWL Time Ontology that we use in this chapter, the classes for representing *events* and *fluents* and their relationships with *obligations*. Subsequently we define one possible example of a domain dependent ontology that will be used in the examples contained in the paper. Then we introduce the part of the ontology that is necessary for representing *events* and the *elapsing of time*. Later on we present the part of the ontology used to represent the content, the condition, the deadline, and the expiration condition of *obligations*. Finally we introduce the part of the ontology and the mechanisms that have to be used to *monitor* the *time evolution* of obligations on the basis of the *actions* and *events* that happen in the system. At the end of this section the graphical representation of the proposed ontology is reported.

4.1 Modelling Time, Events, and Fluents

The first class that has to be introduced is the Agent class that is used to represent the agents involved in the interaction mediated by the open system. Secondly in order to be able to represent obligations with activation and deactivation events correlated to time and with temporal deadlines, we have to find a suitable and efficient way to represent instants and interval of time in the ontology. Given that OWL has not temporal operator, the simplest solution, which pursues also the goal of being interoperable with other ontologies, is to adopt the OWL Time Ontology[5]. Unfortunately the axiomatization of the OWL Time Ontology is very weak and therefore it will be impossible to perform certain type of interesting reasoning on the future evolution of the state of the system. Nevertheless, as we will see in the following subsections, we will try to partially overcome to this problem, in order to be able, at least, to represent and monitor the time evolution of the system. Here we report the list of classes and properties of the OWL Time Ontology that

[5] http://www.w3.org/TR/owl-time/

are relevant for the comprehension of this chapter (they are graphically represented in Figure 1 at the end of this section):

Instant $\sqsubseteq$ TemporalEntity, Interval $\sqsubseteq$ TemporalEntity,
ProperInterval $\sqsubseteq$Interval, TemporalEntity $\equiv$ Instant $\sqcup$ Interval,
hasBeginning: TemporalEntity $\rightarrow$ Instant,
hasEnd: TemporalEntity $\rightarrow$ Instant,
before: TemporalEntity $\rightarrow$ TemporalEntity, InvPro(after,before),
inDateTime: Instant $\rightarrow$ DateTimeDescription,
Discla(ProperInterval,Instant), Instant $\sqsubseteq$ = 1 inDateTime

In order to be able to represent *events* that happen at a certain *instant* of time, or *fluents*, that is, state of affair that holds for a certain *interval* of time, we introduce the class Eventuality and its two subclasses: Event, whose individuals are related to an instant of time, and Fluent whose individuals are related to an interval:

Event $\sqsubseteq$ Eventuality, Fluent $\sqsubseteq$ Eventuality, Discla(Event,Fluent),
atTime: Eventuality $\rightarrow$ TemporalEntity,
Event $\equiv \exists$ atTime.Instant, Fluent $\equiv \exists$ atTime.Interval.

An event is before another event if the first one happens at an instant of time that is before the instant of time of the second one:

evBefore: Eventuality $\rightarrow$ Eventuality,
atTime $\circ$ before $\circ$ atTime^{-} $\sqsubseteq$ evBefore, Tr(evBefore).

Two events that happens at the same instant of time are related by the evSameTime property:

evSameTime: Eventuality $\rightarrow$ Eventuality,
atTime $\circ$ atTime^{-} $\sqsubseteq$ evSameTime, Tr(evSameTime).

Actions are viewed as a particular type of events that have an actor, a recipient and an object:

Action $\sqsubseteq$ Event, hasActor: Action $\rightarrow$ Agent,
hasRecipient: Action $\rightarrow$ Agent, hasObject: Action $\rightarrow$ Object,
Fun(hasActor), Fun(hasRecipient), Fun(hasObject).

Obligations are represented as particular type of *event*: Obligation $\sqsubseteq$ Event, and they are characterized by the event that brings about their creation. Even if, in the common sense perception, obligations are semantically different from events, this choice gives us the flexibility to be able to specify class of actions as content of the obligations and it makes the axiomatization of the notion of obligation fulfilment and violation simpler. An obligation has a *debtor* and a *creditor* as represented by the following properties:

hasDebtor: Obligation $\rightarrow$ Agent, hasCreditor: Obligation $\rightarrow$ Agent
Discla(Obligation,Action).

An obligation has also a content, an activation event, a deactivation event, and a deadline, which are specified using classes, as discussed in Subsection 4.4.

4.2 An Example of a Domain Dependent Ontology

In order to be able to use in the content and in the condition of obligations concrete classes of actions and events, it is necessary to introduce in the ontology domain dependent classes and properties. Those classes have to be subclasses of the class Action or of the class Event. For example we may need to introduce the class of the actions of delivering a certain object to a certain recipient:

$$\text{Deliver} \sqsubseteq \text{Action} \sqcap \exists\ \text{hasRecipient} \sqcap \exists\ \text{hasObject},$$

the class of actions of paying a certain amount of money to a certain recipient:

$$\text{Pay} \sqsubseteq \text{Action} \sqcap \exists\ \text{hasRecipient} \sqcap \exists\ \text{hasObject},$$

and the class of actions of paying by means of a bank transfer, BankTransfer, which is a subclass of the Pay class: $\text{BankTransfer} \sqsubseteq \text{Pay}$. Those classes will be used in Section 4.6 where different types of obligations for the electronic commerce domain will be presented.

For example the action of delivering a book book1 from agent Luca to agent Marco performed at instant1 is described by the following assertions:

Agent(Luca), Agent(Marco), Object(book1), Instant(instant1),
Deliver(deliver1), hasActor(deliver1, Luca), hasRecipient(deliver1,Marco),
hasObject(deliver1,book1), atTime(deliver1,instant1).

4.3 Representing Events, Actions, and the Elapsing of Time

We want to use the specified OWL ontology to represent the evolution in time of the state of the interaction between autonomous and heterogeneous agents in a norm governed framework. This state has to be represented in every software that is in charge of monitoring the behaviour of the interacting agents, a centralized, mixed, or distributed one (the discussion of the advantages and problems due to the choice of one or other architecture is crucial but due to its complexity it is beyond the scope of this specific paper), and may be represented inside the interacting agents in order to let them to reason and plan their future actions on the basis of the rules of the system. It is moreover reasonable that the interacting agents have a partial knowledge of the state of the interaction, which represents only the interaction in which they are involved or that is relevant for the specific agent.

If the system evolution is simulated, the list of events that happen in the system, the list of actions performed by the agents, and the instant of time when they happen, are known at design time and may be initially introduced in the ontology. Differently, if an actual interaction between agents takes place at run-time, it is necessary to tackle two problems. First of all it is required to support agents' communication with an appropriate middle-ware, like for instance the widely used JADE framework[6], or by using web services standard technologies[7]. Regarding the agent communication language (ACL) we plan to adopt the commitment based

[6] http://jade.tilab.com/
[7] http//www.w3.org/standards/webofservices/description

one presented in [9] for the exchanged messages instead of the FIPA-ACL standard semantics[8] that presents a set of well known drawbacks [22]. Secondly it is necessary to write a program in charge of inserting in the ontology a representation of agents' actions and of the events observed, together with the corresponding instant of time when they happened, for example a typical type of action that needs to be recorded in the ontology is the exchange of messages between agents.

Either the interaction is simulated or it is actually happening at run-time, events or actions happen at certain instant of time and it is necessary to state what the temporal relation between those instants is. This can be simply done by asserting which instant comes after another using the after property. Then thanks to the transitivity of the after property, it is possible to deduce the temporal relation that subsists between all instants of time present in the ontology. Alternatively in order to be able to compare two instants of time and assert which one comes after the other the designer may decide to use an external Java program, or an SWRL rule with built-ins for comparisons, or simply inserting the instant of time in the ontology following their temporal order and asserting that the last instant inserted is after the last but one.

Certain subclasses of the class Event are used in the definition of specific obligations as explained in the following sections. In particular it will be certainly necessary to represent at least the following different *types of events*:

- *Time events* are used to represent the events related to the elapsing of time and belong to the TimeEvent $\sqsubseteq$ Event class. This class is disjoint from the Obligation and from the Action classes: Discla(Obligation,TimeEvent), Discla(Action,TimeEvent). A specific time event is related by means of its atTime property to the instant of time when it happens. Notice that a time event actually happens when its instant of time is asserted to belong to the class Elapsed that will be introduced later on.
- *Action events* are used to represent actions performed by the agents, they are represented as individuals of the class Action, for example the action of delivering a product. The action of exchanging a message is a common and very important type of action represented with the class ExchMsg $\sqsubseteq$ Action. It has an actor, the *sender* of the message, a recipient, the *receiver* of the message, an *illocutionary force* (see [9] for more details) connected to the message using the hasForce property whose range is the IllocutionaryForce class, and an object that is the *content* of the message.
- *Change events* are used to represent the events due to the change of the value of a property, they are represented as individuals of the ChangeEvent $\sqsubseteq$ Event class. For example the change in the state of an auction from close to open can be used as condition of the obligation for the auctioneer to declare the current price of the product to be sold. Usually a change event is characterized by the *entity* whose property is changed, the *previous value* and the *subsequent value* of the property, they are all represented as properties of change events. Obviously whenever the performance of an action, or the occurrence of an event,

[8] http://www.fipa.org/specs/fipa00037/SC00037J.pdf

has the effect to change the value of a property of an entity, and if the change event is relevant for one of the obligations represented in the ontology, it is necessary to introduce in the ontology an individual belonging to the ChangeEvent class with a suitable atTime value. This is a fundamental feature of the middle-ware, and it has to be strongly optimized because may be critical in terms of time consuming.

In general when a certain obligation has to be created it may happen that it is necessary to create new subclasses of those classes. Moreover if the new obligation is related to a specific time event (for example the obligation to deliver a book within a given deadline), a new individual, belonging to the TimeEvent class, has to be inserted in the ontology in order to represent such a time event.

In order to model the *elapsing of time* we need to have in the ontology a set of individuals used to represent all the relevant instants of time. An instant of time is *relevant* if an action, or an event, happens at that instant of time, or if such an instant of time is used to create a time event related to the specification of an obligation. The distance between an instant of time and the following one depends on the *time lag* chosen for the system: every type of interaction may have its own reasonable time lag that mainly depends on the frequency on which actions or events happen. During the evolution of the interaction, in order to model the elapsing of time, the individual corresponding to the actual instant of time (of the simulation or of the actual agents interaction) have to be asserted to belong to the $\mathsf{Elapsed} \sqsubseteq \mathsf{Instant}$ class, a special class introduced specifically for this purpose. Every instant of time that is before an elapsed instant of time is itself elapsed as expressed by the following axiom: $\exists \mathsf{before.Elapsed} \sqsubseteq \mathsf{Elapsed}$.

In case the evolution of the system is simulated it is enough to repeatedly assert that the instant of time, subsequent to the current one, is elapsed, and then run the reasoner to deduce all the consequences of the events or actions happened at the current instant of time. Differently if the ontology is used to represent the state of an actual agents interaction, it is necessary to keep aligned the current instant of time represented in the ontology (the last that is asserted to be elapsed) with the external clock, that is, the clock of the world where the agents actually interact. Therefore an instant of time has to be asserted to be elapsed only when its inDateTime property is lower or equal to the time adopted by the interacting agents.

4.4 Representing Specific Obligations

In this chapter we specify how to formalize in the ontology used to represent the state of the interaction among agents their obligations and we describe how to monitor, using semantic web technologies, those obligations. An obligation exists between two specific agents that are the *debtor* and the *creditor* of the obligation. An obligation is characterized by the *instant of time* when the obligation is created, a class of events that may *activate* or *deactivate* it, a *content* described by means of another class, and a *deadline*. We assume (coherently with what is specified in the OCeAN meta-model [9]) that new obligations are created as the effect of the performance of certain communicative acts (like promises), or as consequence of the activation of a norm. A norm is activated whenever an agent, who is interacting with other agents within a certain institutional context, starts to play a role whose behaviour is regulated by the norm. Whenever a new obligation, obl-n, is created at a certain instant of time, instant-n, whose inDateTime property value is equal to the time when the obligation is created (in the following referred as now), the ABox of the ontology has to be automatically updated with the following assertions:

Obligation(obl-n), atTime(obl-n,instant-n), inDateTime(instant-n,now), hasDebtor(obl-n,agent1), hasCreditor(obl-n,agent2).

In addition it is necessary to update the TBox in the following way: the first change consists in defining the specific *activation*, *deactivation*, *content*, and *deadline* classes of the new obligation; secondly it will be necessary to write the axioms for deducing the state of a given obligation, with the goal of monitoring its fulfilment or its violation as described in the following subsection.

The StartEvent-n ⊑ Event class describes the type of events that may activate the obligation obl-n, that is, the conditional event that have to happen in order to make the obligation *activated*. For example in certain electronic commerce scenario an agent may start to be actively obliged to pay a certain amount of money after the reception of the ordered product. Certain obligation may be immediately activated without the need to specify any condition, in this case the StartEvent-n class coincides with the event that create the obligation:

StartEvent-n≡{obl-n}.

If it is possible to deduce that the StartEvent-n class is equal to the empty set ⊥, it means that the obligation obl-n will never be activated. This is a fundamental information for the agents when they are planning their future actions.

The EndEvent-n ⊑ Event class describes the type of events that may expire the obligation, that is, when an expiration event happens the obligation becomes *cancelled* and will not any more become active in the future. The specification of this class is crucial for those obligations that may be activated many times, for example an employer may have the obligation to pay the salary to his/her employees at the end of each month as long as they are employed in the company. Very often the EndEvent-n class is equivalent to the class of the actions that may be used to dismiss an agent from a specific role, the role indicated in the debtor or in the creditor field of the norm that generated the obligation. For example when an agent ceases to be an employer or an employee the obligation to pay the salary becomes cancelled. In some other cases the EndEvent-n class coincides with a fixed deadline, that is, with a certain time event, for example the instant of time when the contract of the employee terminates.

The Content-n ⊑ Action class describe the set of actions whose performance may fulfil or violate the obligation. An crucial aspect of the proposed model is the possibility that an action, belonging to a subclass of the Content-n class, satisfies the obligation. Moreover in the definition of the Content-n class it is also possible to use Boolean class constructors. The union of classes can be used for those cases when either an action belonging to one class or an action belonging to another class may fulfil an obligation.

When an agent has the obligation to perform an action it is necessary to define the deadline (i.e. the instant of time) within which the action has to be performed. For coherence with the other classes we introduce the class Deadline-n ⊑ Timevent even if it contains only one individual: the time event associated to the instant of time that represents the deadline of the obligation. Taking into consideration the existence for every obligation of a start and dead-line event it is natural to introduce a property hasInterval: Obligation→TemporalEntity that binds an obligation to the interval of time within which one action belonging to the Content class has to be performed. Such an interval has a beginning instant of time, an end instant of time, and a duration that can be obtained by means of the hasBeginning, hasEnd, hasDurationDescription properties. The instant of time when the interval of obl-n starts can be deduced on the basis of the instant of time when an individual belonging to the StartEvent-n class happens by introducing the following SWRL rule:

StartEvent-n(?e) ∧ atTime(?e,?inst) ∧ hasInterval(obl-n,?int) →
hasBeginning(?int, ?inst)

The Deadline-n class is equivalent to the class that contains only the time event that happens at the instant of time when the interval of the obligation finishes, as stated in the following axiom:

$\text{Deadline-n} \equiv \exists\, \text{atTime}.(\exists\, \text{hasEnd}^{-}.(\text{hasInterval}^{-} \ni \text{obl-n}))$

It is important to remark that when the deadline of the obligation depends on the instant of time when the obligation is activated, the time event to be used as deadline is unknown when the obligation is created. In this case the Deadline-n class will become defined when the obligation becomes active. A example of this kind of obligations are those obligation where the deadline is equal to the instant of time when the obligation is activated plus a fixed amount of time, for instance the obligation to pay the product within 2 days from its reception. For these type of obligations it is necessary to insert in the ontology also the value of the duration of the interval associated with the obligation. Once the beginning instant and the duration of the interval are known, it is possible to use the following SWRL rule, which uses the swrlb:add built-in, to deduce the value of the end instant of time of the interval (we assume that the duration of the interval is expressed in days):

hasBeginning(?int,?inst1) ∧ inDateTime(?inst1,?dt1) ∧
dayOfYear(?dt1,?day1) ∧ Instant(?int2) ∧ inDateTime(?inst2,?dt2) ∧
dayOfYear(?dt2,?day2) ∧ hasDurationDescription(?int,?d) ∧ days(?d,?value)∧
swrlb:add(?day2,?day1,?value) → hasEnd(?int,?inst2)

For those obligations where the deadline event is a fixed time event that does not depend on the activation event (see for example in Section 4.6 the first type of obligations), it is important to check that the start event happens before the end event. This can be done with the following axiom that has to be written only for obligations whose StartEvent-n and Deadline-n classes are equivalent to a specific time event. In case the deadline time event is before or equal to the start time event the ontology becomes contradictory:

$$\text{Deadline-n} \sqcap (\text{evBefore.StartEvent-n} \sqcup \text{evSameTime.StartEvent-n}) \sqsubseteq \bot$$

In Section 4.6 specific examples will be used to illustrate the definition of the StartEvent, EndEvent, Content, and Deadline classes for different type of obligations.

4.5 Monitoring the State of Obligations

When a new obligation obl-n is created the second change to the TBox consists in introducing the four axioms that are necessary to deduce the state of a given obligation, that is, to deduce if it belongs to the Activated, Cancelled, Fulfilled, or Violated classes. Those classes are subclass of the class Obligation and the Fulfilled and Violated classes are disjoint:

$$\text{Fulfilled} \sqsubseteq \text{Obligation},\ \text{Violated} \sqsubseteq \text{Obligation},\ \text{Activated} \sqsubseteq \text{Obligation},$$
$$\text{Cancelled} \sqsubseteq \text{Obligation},\ \text{DisCla(Fulfilled, Violated)}.$$

The first axiom is the one to deduce that an obligation with a certain StartEvent class is activated. If an event e_s that belongs to the StartEvent-n class of an obligation obl-n happens after or at the same instant of time when the obligation is created, the time at which e_s happens is elapsed, and the obligation has not yet been cancelled, then the obligation becomes activated.

The main problem in writing this axiom is due to the negation that appears in the third condition. OWL reasoners operate under the open world assumption and therefore we cannot simply write in the axiom the condition "*not cancelled*". In fact the conclusion that an obligation is not cancelled can only be reached if the obligation can be definitely proved not to be member of the Cancelled class. To solve this problem we assume that our ABox contains complete information on the events happened or actions performed before the current time of the system. More specifically, we assume to use an external Java program that will always update the ABox whenever an event happens. Moreover we assume that such a program can only insert in the ABox the information that an event is happened at current time *t*, and that it is not possible to insert the information that an event is happened in the past. Starting from these assumptions we can adopt a closed-world perspective on the Cancelled class: an obligation "is not yet been cancelled" if it is not in the Cancelled class. Consequently in order to be able to perform some form of closed world reasoning on the Cancelled class (similarly to the solution proposed in [10]) we introduce in our ontology the explicit closure of such a class. More precisely, we introduce a new class, the $\text{KCancelled} \sqsubseteq \text{Cancelled}$, which is meant to contain all obligations that, at a given time, are known to be in the Cancelled class. To maintain the KCancelled class as the closure of the Cancelled class, we define it periodically as equivalent to the enumeration of all individuals that can be proved to be members of the Cancelled class. This can be done by the external Java program that is also used to update the ABox to keep track of the elapsing of time and of the events that happen in the system. The axiom to deduce if an obligation obl-n is activated is therefore:

Axiom Activated Obl-n:
$\{\text{obl-n}\} \sqcap \neg \text{KCancelled} \sqcap (\exists \text{evBefore}.(\text{StartEvent-n} \sqcap \exists \text{atTime.Elapsed}) \sqcup \exists \text{evSameTime}.(\text{StartEvent-n} \sqcap \exists \text{atTime.Elapsed})) \sqsubseteq \text{Activated}$

An obligation, when is not yet cancelled, may be activated more than once by different start events belonging the StartEvent class. It is important to be able to monitor the time evolution of the obligation for each one of its possible activation event. Therefore we assume that whenever an obligation is activated at instant *i*, an external program has to create a copy of that obligation and associate it to a creation time that is one instant of time later than the instant of time of the current activation *i*. This fact is crucial to avoid that the new copy of the obligation becomes active due to the current activation event.

If an event e_e that belongs to the EndEvent-n class of an obligation obl-n happens after the time when the obligation is created and the time at which e_e happens is elapsed, then the obligation becomes cancelled. It is important to underline that an obligation that is activated may be also cancelled (the Activated and Cancelled classes are not disjoint). This means that it can become fulfilled or violated but also that cannot be any more activated in the future by another start event. For example the obligation to pay the salary to an employee at the end of each month for an entire year becomes cancelled at the end of the year and is activated for twelve times, the last time that the obligation is activated it is also cancelled because the entire year is elapsed. If an end event happens before a start event the obligation is never activated. For example the obligation for a company to keep the streets of a city clear from the snow for a given winter will never be activated if the winter is particularly warm.

Axiom Cancelled Obl-n:
$\{\text{obl-n}\} \sqcap \exists \text{evBefore}.(\text{EndEvent-n} \sqcap \exists \text{atTime.Elapsed}) \sqsubseteq \text{Cancelled}$

As mentioned before the Deadline-n class contains only one time event, the time event within which an action belonging to the Content-n class has to be performed.

If an event e_c that belongs to the Content-n class of an active obligation obl-n (created at i_n) happens at instant i_c, i_c is after or equal to i_n, i_c is before the deadline of obl-n, and i is elapsed, then the obligation becomes fulfilled as expressed by the following axiom.

Axiom Fulfilled Obl-n:
$\{\text{obl-n}\} \sqcap \text{Activated} \sqcap (\exists \text{evBefore}.(\text{Content-n} \sqcap \exists \text{atTime.Elapsed}) \sqcup \exists \text{evSameTime}.(\text{Content-n} \sqcap \exists \text{atTime.Elapsed})) \sqcap \exists \text{evBefore}.(\text{Content-n} \sqcap \exists \text{evBefore.Deadline-n}) \sqsubseteq \text{Fulfilled}$

If the time event that represents the deadline of an active obligation obl-n elapses and the obligation is not yet fulfilled, the obligation has to become violated. Similarly to what we did for writing the axiom for the activation of obligations, in order to write the axiom to deduce that an obligation is cancelled we need to introduce the explicit closure of the Fulfilled class: the class KFulfilled ⊑ Fulfilled. The KFulfilled class is meant to contain all obligations that, at a given time, are known to be in the Fulfilled class. To maintain the KFulfilled class updated we define it periodically, by means of the external program, as equivalent to the enumeration of all individuals that can be proved to be members of the Fulfilled class. The axiom to deduce that an obligation obl-n is violated is:

Axiom Violated Obl-n:
{obl-n} ⊓ Activated ⊓ ¬ KFulfilled ⊓
∃evBefore.(Deadline-n ⊓ ∃ atTime.Elapsed) ⊑ Violated

Initially KCancelled ≡ KActivated ≡ KFulfilled ≡ KViolated ≡ Nothing then a Java external program has to update their extension on the basis of the deductions of the reasoner. In Figure 1 the graphical representation of the classes and properties introduced in the previous sections is depicted.

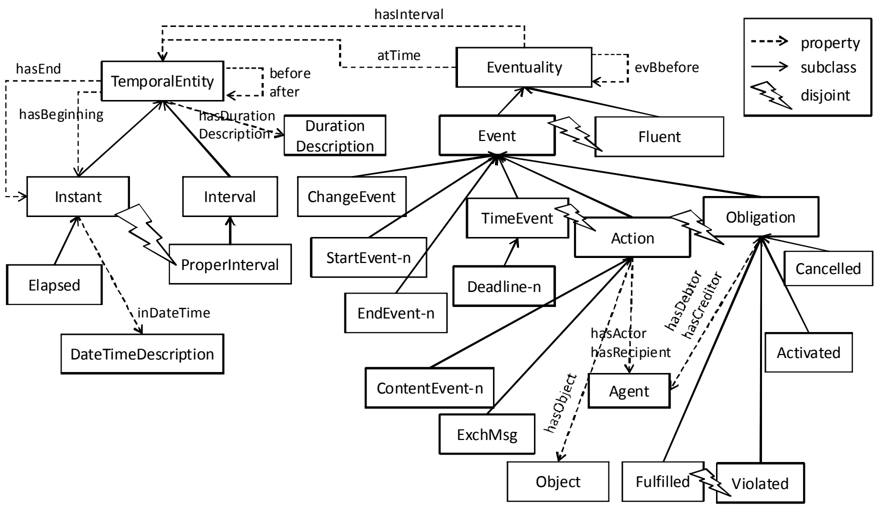

Fig. 1. Graphical representation of the ontology. Properties are represented with dotted lines, solid lines are used for subclasses.

4.6 Possible Type of Obligations

A first type of obligations are those obligations whose StartEvent and Deadline classes are equivalent to a specific time event. It means that the obliged action described with the Content class has to be performed between two specific instants of time. An example of an obligation of this type is the obligation obl-1 created at instant1 from agent Marco to agent Luca to pay 5 euro between instant of time instant2 and instant4 having certain specific dates as inDateTime properties.

To model the obligation obl-1 it is necessary to add to the ABox the following assertions:

Obligation(obl-1), Agent(Marco), Agent(Luca), Thing(5euro),Instant(instant1), atTime(obl-1,instant1),hasDebtor(obl-1,Marco), hasCreditor(obl-1,Luca), ProperInterval(interval1), hasInterval(obl-1,interval1), hasEnd(interval1,instant4),TimeEvent(tevent4), Instant(instant4), atTime(tevent4,instant4),

For this kind of obligations the StartEvent-1 classes consist of only one element: the time event that happens at instant2:

TimeEvent(tevent2), Instant(instant2), atTime(tevent2,instant2),
after(instant2,instant1), after(instant4,instant2),
StartEvent-1 $\equiv$ {tevent2},
Content-1$\equiv$Pay $\sqcap$ hasActor$\ni$Marco $\sqcap$ hasRecipient$\ni$Luca $\sqcap$ hasObject$\ni$5euro.

The four axioms for deducing the state of obligations contextualized to this specific obligation have to be inserted in the ontology. Given that this obligation can become active only one time, it is not interesting to define the EndEvent class.

A crucial aspect of the proposed approach is that it is more flexible than other ones, in fact, given that the content of the obligations is expressed using a class of possible actions, the interacting agents have the flexibility to choose which one to perform. Moreover, if an event that belongs to one of the subclasses of the Content class happens, the obligation may equally become fulfilled. For example, if the bank transfer event (represented with the individual bankTr1$\in$BankTransfer where BankTransfer $\sqsubseteq$ Pay) from Marco to Luca of an amount of 5 euro happens after the activation event and before the deadline event, the obligation obl-1 becomes fulfilled.

The content of an obligation could also be the performance of either one class or another class of actions. This type of Content class can be represented using the union of two or more classes of actions. For example the obligation from Marco to Luca to either pay 5 euro to Luca or donate 6 euro to Unicef between instant2 and instant4 is identical to the previous obligation except for the Content-1 class that becomes:

Content-1 $\equiv$
(Pay $\sqcap$ hasActor$\ni$Marco $\sqcap$ hasRecipient$\ni$Luca $\sqcap$ hasObject$\ni$5euro) $\sqcup$
(Pay $\sqcap$ hasActor$\ni$Marco $\sqcap$ hasRecipient$\ni$Unicef $\sqcap$ hasObject$\ni$6euro)

A second type of obligations has the StartEvent class that can be interpreted as a condition for the activation of the obligation (a conditional obligation) and whose Deadline class depends on the time of its activation. An example of an obligation of this type is the obligation obl-2 created at instant1 from agent Marco to agent Luca to pay 5 euro within 2 days from the reception of the book (book1) on condition that the book was delivered from Luca to Marco. Besides the assertions previously introduce we have to add in the ABox those ones:

Obligation(obl-2), atTime(obl-2,instant1),
hasDebtor(obl-2,Marco), hasCreditor(obl-2,Luca), Object(book1),
ProperInterval(interval2), hasInterval(obl-2, interval2),
hasDurationDescription(interval2,duration2), days(duration2, 2),
StartEvent-2 ≡ Deliver ⊓ hasActor∋Luca ⊓ hasRecipient∋Marco ⊓
hasObject∋book1,
Content-2 ≡Pay ⊓ hasActor∋Marco ⊓ hasRecipient∋Luca ⊓ hasObject∋5euro,
EndEvent-2 ≡ {teventk}.

Reasonably the EndEvent-2 class is equivalent to the time event teventk whose instant property can be calculated as the time of creation of the obligation plus 3 months. This means that if the book is not delivered within 3 months Marco is not any more conditional obligated to pay for the book after its reception. As usual the four axioms presented in the previous section for deducing the state of an obligation, contextualized to this specific obligation, have to be inserted in the ontology.

A third type of obligations has not condition, that is, their StartEvent class is equivalent to the time of the creation of the obligation. Due to this fact the deadline of this type of obligations can be set when the obligation is created on the basis of the duration of the interval. An example of an obligation of this type is the obligation obl-3 created at instant1 from Marco to Luca to pay 5 euro before tomorrow, where tomorrow is computed at the creation of the obligation to be represented by the instant of time instant4. This obligation can be represented with the following assertions and axioms:

Obligation(obl-3), atTime(obl-3,instant1), hasDebtor(obl-3,Marco),
hasCreditor(obl-3,Luca) TimeEvent(tevent1), atTime(tevent1,instant1)
ProperInterval(interval3), hasInterval(obl-3,interval3),
hasEnd(interval3,instant4),
StartEvent-3 ≡ {tevent1}
Content-3 ≡ Pay ⊓ hasActor∋Marco ⊓ hasRecipient∋Luca ⊓ hasObject∋5euro

As already explained the four axioms for deducing the state of this obligation have to be inserted in the ontology.

5 A Case Study: Obligations in Vehicle Repair Contracts

In this Section we formalize and monitor the vehicle repair contract described in [16] using the model presented in this chapter. The scenario is as follows: a *repair contract* regulates the interactions between a client agent called cl and a vehicle repair company, called rc. A repair contract specifies details concerning a particular repair. The interaction between cl and rc is described as follows: when rc receives a request from cl to undertake a repair job, it has to send a repair contract within *x* days. In response, cl sends an acceptance or rejection message within *y* days. If accepted, cl has to send the vehicle within *k1* day from the acceptance. rc then waits for the vehicle to arrive, failing which it sends two reminders to cl. If the vehicle fails to arrive, it takes an offline action. As per the contract, if the vehicle arrives rc is obliged to assess the damage, repair the vehicle, and send a report to cl within *k2* days from the reception of the vehicle. On receiving the report, cl is obliged to send payment to rc within *k3* days from the reception of the report. If the payment is not sent, rc sends two reminders to cl and then takes an offline action. If the payment is sent cl has to pick-up the vehicle within *k4* days from the reception of the report.

Every action has to be performed within a certain number of days, and the actual deadline is computed on the basis of the time when a certain event happens, the maximum duration of each activity is defined in the contract and may vary from one contract to another. Almost all these obligations are conditional obligations with deadline computed on the basis of the time of their activation; therefore they are similar to the second type of obligations presented in Section 4.6. Initially the interaction between rc and cl is devoted to the definition of the properties of a specific repair contract that is characterized by the type of the repair, the price, four duration of time used to compute the deadlines of the obligations for agent cl, and two duration of time used to compute the deadlines of the obligations for agent rc. We represent such a contract as an individual of the class VehicleRepairContract having the properties hasRepairType, hasPrice, hasDuration1,..., hasDuration6. This is another example of a domain dependent ontology. If the contract is accepted by both parties six conditional obligations start to hold, four for agent cl and two for agent rc. Subsequently the interaction is devoted to the execution of the contract. Given that the interacting agents belong to different owners having different interests, their behaviour has to be monitored to verify its compliance with the obligations.

In order to define the contract and reach an agreement on the value of the properties used to characterize the contract the two agents need to interact at least two times, but can interact also more times. A contract is *complete* if all its properties are set and therefore it belongs to the CompleteContract ⊑ VehicleRepairContract class as stated by the following axiom:

CompleteContract ≡ ∃ hasRepairType.TypeRepair ⊓ ∃ hasPrice ⊓
∃ hasDuration1 ⊓ ... ⊓ ∃ hasDuration6

The contract definition phase is regulated by two obligations: once is the obligation for rc to send a complete contract to agent cl within *x* days from the reception of the request from cl; the second is the obligation for agent cl to accept or reject a complete contract offer within *y* days. In case cl rejects the proposed contract the negotiation can continue with new requests and counter offers on the basis of the pro-activity of the two involved agents. If agent cl accepts the proposed contract then six new conditional obligations are created having as interval the duration specified in the contract. The first obligation for rc can be represented as:

Obligation(obl-4), atTime(obl-4,instant1), Instant(instant1),
ProperInterval(interval4), hasInterval(obl-4, interval4),
hasDurationDescription(interval4,duration4), days(duration4, x),
StartEvent-4 ≡ ExchMsg ⊓ hasActor∋cl ⊓ hasRecipient∋rc ⊓
hasForce∋request ⊓ ∃ hasObject.VehicleRepairContract.

The Content-4 class contains the actions of sending a request message from agent rc to agent cl with as content an individual belonging to the CompleteContract class:

Content-4 ≡ ExchMsg ⊓ hasActor∋rc ⊓ hasRecipient∋cl ⊓
hasForce∋request ⊓ hasObject.CompleteContract.

The obligation for agent cl to accept or reject a complete contract offer within *y* days can be represented as:

Obligation(obl-5), atTime(obl-5,instant1), Instant(instant1),
ProperInterval(interval5), hasInterval(obl-5, interval5),
hasDurationDescription(interval5,duration5), days(duration5, y),
StartEvent-5 ≡ Content-4.

The Content-5 class contains the actions of accepting or rejecting the contract whose proposal activated the obligation obl-5:

Content-5 ≡ (ExchMsg ⊓ hasActor∋cl ⊓ hasRecipient∋rc ⊓
hasForce∋accept ⊓ ∃ hasObject.(∃ hasObject^{-} StartEvent-5)) ⊔
(ExchMsg ⊓ hasActor∋cl ⊓ hasRecipient∋rc ⊓ hasForce∋reject ⊓
∃ hasObject.(∃ hasObject^{-} StartEvent-5))

Due to space limitation we will not describe in detail the formalization of all the other conditional obligations that will be created once the contract is accepted, they are similar to the second type obligations introduced in section 4.6. The application independent ontology described in this chapter with an ABox that contains the obligations described in the previous sections can be downloaded from the author's web page[9].

6 Conclusions and Future Works

In this chapter we presented a formal model for the specification and monitoring, using semantic web technology, of obligations whose content is a class of possible action, with activation and deactivation event and with deadline. The main goal of having this type of formal specification of obligations is to be able to have more flexible interactions among autonomous agents. This is possible because agents can decide at run-time which is the best action, among the ones belonging to the Content class, to perform in order to fulfil their obligations. This work is a first step in the broader project of formalizing, using semantic web technology, also prohibitions and permissions that present some crucial differences with respect to obligations. Another very important aspect of the formalization of normative concepts in open system is, besides their monitoring as explained in this chapter, their enforcement by the definition of sanctions and recovery actions.

Another interesting problem would be the definition of constrains for the validation of a normative specification and the introduction of mechanism for early

[9] http://www.people.lu.unisi.ch/fornaran/ontology/ObligationsOntology.html

detection of problematic situations. For example being able to point out that an agent is at the same time obliged to perform an action and obliged to perform another action that is inconsistent with the first one, like being in two different places at the same time. Another very interesting open problem is being able to demonstrate that a given set of obligations has some soundness properties [17].

Finally regarding the decision to adopt semantic web technology as formal language, there is still the open problem of better understanding what part of the model it is better and possible to represent in the ontology in order to be able to reason on it and what part of the model it is better to represent in the external application because current semantic web standards do not support its representation.

Acknowledgments. We would like to thank Marco Colombetti for the interesting discussions concerning this work.

References

1. Arcos, J.L., Esteva, M., Noriega, P., Rodríguez-Aguilar, J.A., Sierra, C.: Engineering open environments with electronic institutions. Engineering applications of artificial intelligence 18(2), 191–204 (2005)
2. Artikis, A., Sergot, M., Pitt, J.: Animated Specifications of Computational Societies. In: Castelfranchi, C., Johnson, W.L. (eds.) Proceedings of the 1st International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2002), pp. 1053–1061. ACM Press, New York (2002)
3. Boella, G., Noriega, P., Pigozzi, G., Verhagen, H. (eds.): Normative Multi-Agent Systems, Dagstuhl, Germany. Dagstuhl Seminar Proceedings, vol. 09121. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany (2009)
4. da Silva, V.T.: 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), 113–155 (2008)
5. Dastani, M., Grossi, D., Meyer, J.-J., Tinnemeier, N.: Normative multi-agent programs and their logics. In: Boella, G., Noriega, P., Pigozzi, G., Verhagen, H. (eds.) Normative Multi-Agent Systems, Dagstuhl, Germany. Dagstuhl Seminar Proceedings, vol. 09121, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany (2009)
6. Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall PTR, Upper Saddle River (2005)
7. Faci, N., Modgil, S., Oren, N., Meneguzzi, F., Miles, S., Luck, M.: Towards a monitoring framework for agent-based contract systems. In: Klusch, M., Pěchouček, M., Polleres, A. (eds.) CIA 2008. LNCS (LNAI), vol. 5180, pp. 292–305. Springer, Heidelberg (2008)
8. Fornara, N., Colombetti, M.: Operational specification of a commitment-based agent communication language. In: Castelfranchi, C., Johnson, W.L. (eds.) Proceedings of the 1st International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2002), pp. 535–542. ACM Press, New York (2002)
9. Fornara, N., Colombetti, M.: Specifying Artificial Institutions in the Event Calculus. In: Handbook of Research on Multi-Agent Systems: Semantics and Dynamics of Organizational Models, Information science reference, ch. XIV, pp. 335–366. IGI Global (2009)

10. Fornara, N., Colombetti, M.: Ontology and time evolution of obligations and prohibitions using semantic web technology. In: Baldoni, M., Bentahar, J., van Riemsdijk, M.B., Lloyd, J. (eds.) DALT 2009. LNCS, vol. 5948, pp. 101–118. Springer, Heidelberg (2010)
11. Fornara, N., Viganò, F., Colombetti, M.: Agent communication and artificial institutions. Autonomous Agents and Multi-Agent Systems 14(2), 121–142 (2007)
12. Fornara, N., Viganò, F., Verdicchio, M., Colombetti, M.: Artificial institutions: A model of institutional reality for open multiagent systems. Artificial Intelligence and Law 16(1), 89–105 (2008)
13. García-Camino, A., Rodríguez-Aguilar, J.A., Sierra, C., Vasconcelos, W.: Constraint rule-based programming of norms for electronic institutions. Autonomous Agents and Multi-Agent Systems 18(1), 186–217 (2009)
14. Governatori, G., Rotolo, A.: How do agents comply with norms? In: Boella, G., Noriega, P., Pigozzi, G., Verhagen, H. (eds.) Normative Multi-Agent Systems, Dagstuhl, Germany. Dagstuhl Seminar Proceedings, vol. 09121, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany (2009)
15. Kagal, L., Hendler, J., Berners-Lee, T.: Introduction. In: Web Semantics: Science, Services and Agents on the World Wide Web, vol. 7(1), pp. vii–ix (2009); The Semantic Web and Policy
16. Lomuscio, A., Penczek, W., Solanki, M., Szreter, M.: Runtime monitoring of contract regulated web services (extended abstract). In: Proceedings of the 9th International Conference on Autonomous Agents and Multi-Agent systems (AAMAS 2010), Toronto, Canada, pp. 1449–1450. ACM, New York (2010)
17. Singh, M.P., Chopra, A.K.: Correctness properties for multiagent systems. In: Baldoni, M., Bentahar, J., van Riemsdijk, M.B., Lloyd, J. (eds.) DALT 2009. LNCS, vol. 5948, pp. 192–207. Springer, Heidelberg (2010)
18. Uszok, A., Bradshaw, J.M., Lott, J., Breedy, M., Bunch, L., Feltovich, P., Johnson, M., Jung, H.: New developments in ontology-based policy management: Increasing the practicality and comprehensiveness of KAoS. In: IEEE International Workshop on Policies for Distributed Systems and Networks, vol. 0, pp. 145–152 (2008)
19. van der Torre, G.E.L., Boella, G., Verhagen, H. (eds.): Special Issue on Normative Multiagent Systems. Autonomous Agents and Multi-Agent Systems, vol. 17. Springer, Netherlands (August 2008)
20. Vázquez-Salceda, J., Dignum, V., Dignum, F.: Organizing multiagent systems. Autonomous Agents and Multi-Agent Systems 11(3), 307–360 (2005)
21. Weske, M.: Business. In: Process Management Concepts, Languages, Architectures. Springer, Heidelberg (2008)
22. Wooldridge, M.: Verifiable semantics for agent communication languages. In: Demazeau, Y. (ed.) Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS 1998), Washington, DC, USA. IEEE Computer Society, Los Alamitos (1998)
23. Xu, L.: A Framework for E-markets: Monitoring Contract Fulfillment. In: Bussler, C.J., Fensel, D., Orlowska, M.E., Yang, J. (eds.) WES 2003. LNCS, vol. 3095, pp. 51–61. Springer, Heidelberg (2004)
24. Yolum, P., Singh, M.: Reasoning about commitment in the event calculus: An approach for specifying and executing protocols. Annals of Mathematics and Artificial Intelligence 42, 227–253 (2004)