SIMPLE: a Language for the Specification of Protocols, Similar to Natural Language

Dave de Jonge and Carles Sierra

IIIA-CSIC, Bellaterra, Catalonia, Spain
{davedejonge,sierra}@iiia.csic.es

Abstract. Large and open societies of agents require regulation, and therefore many tools have been developed that enable the definition and enforcement of rules on multiagent systems. Unfortunately, most of them are designed to be used by computer scientists and are not suitable for normal users with average computer skills. Since more and more tools nowadays are running as cloud services accessible to anyone (e.g. Massive Open Online Courses and social networks), we feel there is a need for a simple tool that allows ordinary people to create rules and protocols for these kinds of environments. In this paper we present ongoing work on the development of a new programming language for the definition of protocols for multiagent systems, which is so simple that anyone should be able to use it. Although its syntax is strict, it looks very similar to natural language so that protocols written in this language can be understood directly by anyone, without having to learn the language beforehand. Moreover, we have implemented an easy to use editor that helps users writing sentences that obey the syntax rules, and we have implemented an interpreter that can parse such protocols and verify whether they are violated or not.

1 Introduction

In open multiagent systems (MAS) where any agent can enter and leave at will and the origin of the agents is unknown one needs a mechanism to regulate the behavior of those agents. Just like in human societies, rules need to be imposed in order to prevent the agents from misbehaving and abusing system resources. A good example is that of an auction taking place under a specific protocol. An English auction protocol for example, requires the buyers to make increasing bids, and stops when the auctioneer says so, after which the buyer with the highest bid wins the auction. In a Dutch auction on the other hand, bids are decreasing, and the first buyer to accept a bid wins the auction.

Many systems for the implementation of such regulatory systems have been developed, such as ANTE [6], MANET [27], S-MOISE+ [16], and EIDE [11]. They allow users to define a set of rules and then impose those rules on the agents in a MAS (the term 'agents' may here refer to software agents as well as to human beings). This enforcement of rules may happen either by punishing misbehaving agents, or by simply making it impossible to violate them, which is called *regimentation*.

One common characteristic of these systems is that they are mainly designed with computer scientists as their target users. They require knowledge of multiagent systems,

programming languages and / or formal logic. For people with no more than average computer skills they are unfortunately too complicated.

We expect however that agent technologies will become more and more common in the near future, creating a demand for simple tools to maintain and organize such systems and that can be used by ordinary people. We can compare this for example with the evolution of web development. In the early days of the Internet, developing a web page was considered an advanced task that would only be undertaken by computer experts, and hence web development languages such as HTML, PHP and SQL were developed to be used by professional programmers. However, as web pages became more and more abundant and every shop, social club, or sports team wanted to have its own web page, many tools such as DreamWeaver and WordPress were introduced to make the creation of web pages a much simpler task. We strive for a similarly easy tool for the development of multiagent systems.

A good example of where such a tool would be useful is the organization of online classes, because teachers often want to put restrictions on their students. Teachers may for example require that students only take a certain exam after they have passed all previous exams. In this way teachers make sure they do not waste their time correcting exams of students that do not study seriously anyway. Another example could be the process of organizing a conference, where one requires authors to submit before a deadline, or one requires the program chair to appoint at least 3 reviewers to each paper. Also, one can think of a tool that allows users to set up their own social networks, with their own specific rules, as suggested in [17].

Therefore, in this paper we present ongoing work on the development of a new language to define protocols for multiagent systems. This language is so close to natural language that it can be understood directly by anyone without prior knowledge of any other programming language. We call this language SIMPLE, which stands for SIMple Protocol LanguagE. Although it *looks* very similar to natural language, it has in fact a strict syntax. Together with this language we also present two tools: an editor that makes it very easy for users to write well-formed sentences, and an interpreter that parses the source file and makes sure that the rules defined in it are indeed enforced. The fact that the language comes with an editor is very important, because it enables the users to write correct protocols without having to know the rules of the language by heart. In fact, it even makes it impossible to write syntactically incorrect sentences.

We would like to stress that this language is not meant to program the agents themselves. It is only meant to program the organizational structure between the agents. That is: it puts restrictions on the agents in their actions, but does not dictate entirely what they ought to do; the agents still have the freedom to make autonomous decisions, as long as these decisions comply with the protocol. The protocol written in this language does not specify what the agents *must* do, but only specifies what the agents *can* do.

We have developed SIMPLE according to the following guidelines:

- The language should stay as close as possible to natural language.
- The syntax should remain strict: sentences must be well formed, and every well formed sentence can only have one correct interpretation.
- Given a protocol written in this language anyone should immediately be able to understand what it means, even if he or she has never seen our language before.

– Users should be able to write a protocol in this language without having to spend any time learning the language.

The only thing we require from the user is that he or she be familiar with the English language. The language as presented here is only the very first version, and we plan to extend it much further in the future.

The rest of this paper is organized as follows: in Section 2 we give a short overview of previous work done in this field. Next, in Section 3 we explain the assumptions that we have made about the set-up of any MAS to which our language is applied. In Section 4 we describe the syntax rules of our language. Next, in Section 5 we explain how our interpreter parses text files written in our language and enforces its rules upon the agents. Then, in Section 6 we give two examples of protocols written in SIMPLE, and for which we have tested that they are successfully parsed and enforced by our interpreter. And finally, in Section 7 we describe the further extensions that we are planning to add to our language.

2 Related Work

Regulatory systems have been subject of research for a long time and a number of frameworks have been implemented that often consist of tools for implementing, testing, running and visualizing protocols. Examples of such frameworks are ANTE [6], MANET [27], S-MOISE+ [16], and EIDE [11]. A comparative study of some of those systems has been made in [12].

ANTE [6] has been implemented as a JADE-based platform, including a set of agents that provide contracting services. It integrates automatic negotiation, trust & reputation and Normative Environments. Users and agents can specify their needs and indicate the contract types to be created. Norms governing specific contract types are predefined in the normative environment. Although ANTE has been targeting the domain of electronic contracting, it was conceived as a more general framework having in mind a wider range of applications.

The MANET [27] meta-model is based on the assumption that the agent environment is composed of two fundamental building blocks: the physical environment, concerned with agent interaction with physical resources and with the MAS infrastructure, and the social environment, concerned with the social interactions of the agents. In the MANET meta-model it is assumed that the normative system can be composed of three structural components: agents, objects and spaces.

In the EIDE framework agents interact with each other in a so called *Electronic Institution*. The agents are grouped in to conversations, which are called *Scenes*. The institution has a specification that defines how agents can move from one scene to another and defines a protocol for each scene. Within a scene the agents interact by sending messages to one another. Each agent in the system has a special agent assigned to it, called its Governor, which checks whether the messages sent by the agent satisfy the protocol, and blocks them when they do not. The EIDE framework comes with a graphical tool called *Islander* [10] that allows people to create institution specifications in a visual manner. Protocols in Islander are represented as finite state machines, drawn

as a graph in which the states are the vertices and the state-transitions are the edges. Every message sent triggers a state transition.

In order to define rules and norms for multiagent systems, a vast amount of languages and logics have been proposed. It would be impossible to list all the relevant work in this field here, so we just mention some of the most important examples. A logical system to define norms and rules is called a *deontic* logic. The best known system of deontic logic is called Standard Deontic Logic (SDL) [30]. Important refinements of this logic are Dyadic Deontic Logic (DDL) [20] and Defeasible Deontic Logic [25]. Furthermore, an extension of this taking temporal considerations into account was proposed in [14]. In [22] A system to formalize norms using input/output logic was proposed, while in [15] the authors provide a model for the formalization of social law by means of Alternating-time Temporal Logic (ATL). In [19] the author proposes the use of Linear Time Logic (LTL) to express norms. Other important approaches are based on Propositional Dynamic Logic (PDL) [23], on See-to-it-that logic (STIT) [4] and on Computational Tree Logic (CTL) [5]. Models for the verification of expectations in normative systems are proposed in [8] and [1], and in [26] the authors introduce the *nC+* language for representing normative systems as state transition systems.

The above mentioned systems however mainly focus on the theoretical properties of regulatory systems. Work that is more focused on the actual implementation of such systems is for example [21] which proposes a model to define rules in the Z language, while in [3] the authors propose the use of Event Calculus for the specification of protocols. A programming language designed to program organizations, called 2OPL, was introduced in [9]. Other important examples of languages and frameworks for the implementation of norms and rules are described in: [29], [2], [28], [13], [18], and [7].

Although some of the above mentioned languages are more user friendly than others, it still seems that they all require the user to be a computer scientist or at least has some knowledge of programming, logic or mathematics. To the best of our knowledge no work has been published on the specification of protocols that aims for truly inexperienced users and tries to stay as close as possible to natural language.

There do exist a number of programming languages that claim to be similar to natural language such as hyperTalk[1] and PlainEnglish[2], but most of them still aim at real programmers, albeit that they aim for *beginning* programmers. The only exception that we know of, is a language called Inform 7 [24]. This is a language that in many cases truly reads like natural language, but the main difference with SIMPLE is that it is developed for an entirely different domain. Inform 7 is a language to write *Interactive Fiction*: an art form that lies somewhere in between literature and computer games.

We think that one of the main reasons that Inform 7 can stay very close to natural language, is that it is highly adapted to a very specific domain. This restricts the possible things a programmer may want to express and hence keeps the language manageable. We have taken a similar approach: our language is only intended to be used as a language for implementing protocols for multiagent systems, and although it could possibly be useful for other domains too, we restrict our attention to this domain.

[1] `http://en.wikipedia.org/wiki/HyperTalk`
[2] `http://www.osmosian.com`

3 Basic Ideas

We assume a multiagent system in which agents exchange messages according to some given protocol. These agents may be autonomous software agents, or may be humans, acting through a graphic user interface. The agents are however not in direct contact with one another. Every message any agent sends first passes a central server that verifies whether the message satisfies the protocol. If a message does not satisfy the protocol, then it is blocked by the server and it will not arrive at its recipients. Note that this is a form of regimentation. In this paper we will not consider any forms of punishment, and assume protocols are only enforced by means of regimentation. We assume that the life-cycle of the MAS is as follows:

1. A user (the *protocol designer*) writes a protocol in our language and stores it in a text file.
2. He or she launches a communication server, with the location of the text file as a parameter.
3. The interpreter, which is part of the server application, parses the text file.
4. Agents connect to the server through a TCP/IP connection and send messages to one another.
5. Every such message is checked by the interpreter. If it does not satisfy the protocol, it is blocked. If it does satisfy the protocol it is forwarded to its intended recipients.
6. The agent that intended to send the message is notified by the server whether the message has been delivered correctly or not.

The text file is not compiled, but is directly parsed by the interpreter, so the language is human-readable and machine-readable at the same time.

Protocols written in SIMPLE have a closed-world interpretation: every message is considered illegal by default, unless the protocol specifies that it is legal. In order to determine which messages are legal, we use a system based on the notion of 'rights' and 'events', meaning that an agent obtains the right to send a specific message if a certain event has (or has not) taken place. The assignment of such rights is determined by if-then rules in the protocol.

We currently assume agents can send messages following one of these two patterns:

- ('say', `x`)
- ('tell', `y`, `z`)

in which the sender can replace `x`, `y` and `z` by any character string (we will see later that the 'tell' message has the interpretation that the value filled in for `z` will be assigned to a variable of which the name is the string filled in for `y`). The current version of the language does not yet allow users to specify the recipient of a message, so for now we assume that any message is always sent to all the other agents in the MAS. We plan this to change in future versions of SIMPLE. Also, we expect that future versions will support more types of messages.

The interpreter keeps a list of **rights** for each agent in the MAS. A right is a tuple of one of the two following forms:

- ('say', `v`)

– (‘tell’, w)

We say that a right (‘say’, v) **matches** a message (‘say’, x) if and only if x is equal to v, or v is the key word ‘anything’. A right (‘tell’, w) matches a message (‘tell’, y, z) if and only if y equals w. For example: if the agent has the right (‘tell’, ‘price’) then it matches the message (‘tell’, ‘price’, ‘$100’). A message is considered legal if the agent sending the message has at least one right that matches the message. Whenever the interpreter determines that a message is legal, it stores a copy of that message, together with the name of its sender, in the interpreter’s **event history**.

One concept that we have borrowed from EIDE is the concept of a *role*. The rules in the protocol never refer to specific individuals, because we assume that at design time the designer cannot know which agents are going to join the MAS at run time. Instead, the protocol assigns rights to agents based on the roles they are playing. Every agent that enters the MAS (i.e. connects to the communication server) must choose a specific role to adopt, from a number of roles that are defined in the protocol. An auction protocol for example, could define the roles *buyer* and *auctioneer*. The protocol could then define a rule saying that a buyer can only make a bid after the auctioneer has opened the auction.

4 Description of the Language

A protocol is written as a set of sentences that look like natural language, but follow a strict syntax. Although in this paper we will often start sentences with a capital, this is not necessary, as the language is entirely case-insensitive. Like in natural language, the end of a sentence is marked with a period. Unlike most other programming languages, variable names are allowed to contain spaces. Another important property of this language, as we will see at the end of this section, is that it is impossible to write inconsistent protocols.

Definition 1. *A **role definition sentence** is a sentence of the form:*

This protocol defines the role r1 *(plural:* r2*).*

Where the protocol designer can replace r1 *and* r2 *by any character string. The string* r1 *is called the **singular role name** and* r2 *is called **plural role name**.*

For each role in the protocol there must also be exactly one such role definition sentence. For example:

This protocol defines the role buyer (plural:buyers).

Definition 2. *A **role constraint sentence** is a sentence of one of the following forms:*

– *There can be any number of* r.
– *There must be at least* x r.
– *There can be at most* x r
– *There must be at least* y *and at most* x r.
– *There must be exactly* x r.

Where x *and* y *can be any positive integer with* y < x *and* r *is a plural role name from one of the role definition sentences, except in the case that* x=1*, in which case* r *must be a singular role name.*

The following sentence is an examples of a role constraint sentence:

There must be at least 2 buyers.

For each role in the protocol there must be exactly one such role constraint sentence. The interpreter makes sure that these role constraints are not violated. That is, when an agent tries to connect to the communication server with a role for which there are already too many participants, the connection will be refused. If there are not yet enough participants for every role, then every message is considered illegal. In other words: the agents can only start sending messages to one another when there are enough participants for every role.

The main idea of the language, as explained above, is that rights are assigned to the agents by means of if-then rules. An example of such a rule could be:

If the auctioneer has said 'open' then any buyer can tell his bid price.

In order to precisely define which sentences are well formed we first need to introduce a number of terms, namely: *quantifiers*, *identifiers*, *conditions*, and *consequences*.

Definition 3. *A **quantifier** is any of these keywords: no, any, every, a, an, the, that*

Definition 4. *An **identifier** is a sequence of characters of one of the following forms:*

- q r
- *no one*
- *anyone*
- *everyone*
- *he*

Where q *can be any quantifier and* r *can be any singular role name. Identifiers of the form* no r *as well as the identifier 'no one' are called **negative identifiers**. All other identifiers are called **positive identifiers**.*

Definition 5. *A **past-event condition** is a string of characters of one of the following forms:*

- id *has said '*x*'*
- id *has told* x
- pid *has not said '*x*'*
- pid *has not told* x

where id *can be any identifier and* x *can be any character string, and* pid *can be any positive identifier. A past-event condition is called negative if it contains the keyword 'not' or if it contains a negative identifier. A past-event condition is called positive otherwise.*

A past-event conditions is a specific type of condition. Other types of condition are defined later. The idea behind this is that a positive past-event condition is considered true if there is any message in the event history that matches the condition. For example the condition *any buyer has said 'hello'* is considered true if there exists a message in the event history of the form ('say', 'hello') which was sent by an agent playing the role *buyer*. A negative past-event condition is considered true if there is no message in the event history that matches the condition.

Definition 6. *A **right-update consequence** is a string of characters of one of the following forms:*

- `pid` *can say* '`x`'
- `pid` *can tell* `x`

where `pid` *can be any positive identifier and* `x` *can be any character string.*

A right-update consequence is a specific type of consequence. Other types of consequences are defined later on.

We can now construct sentences ('rules') of the form *If A then B*, where A is a conjunction of conditions and B is a conjunction of right-update consequences. We say that a rule is **active** if all its conditions are true. Then the idea is that an agent has the right to send a specific message if and only if there is an active rule with right-update consequence that matches that message.

Identifiers are used inside conditions and consequences to determine to which set of agents these conditions and consequences apply. We would like to remark that the quantifiers 'a', 'an', 'any' and 'the' all have exactly the same meaning, so the language contains some redundancy. However, we do consider it very useful to have all of them in the language because they help the protocol designer to write more natural sentences. For example, if an auction protocol contains only one auctioneer it makes much more sense to talk about 'the auctioneer' than about 'any auctioneer'.

Also note that we have included the quantifier 'that'. This quantifier refers to any agent that was also referred to by the last quantifier earlier in the sentence. For example, suppose that a buyer called Alice says 'hello' and then a buyer called Bob says 'hi', then the condition:

any buyer has said 'hello' and any buyer has said 'hi'

is true. However, the condition:

any buyer has said 'hello' and that buyer has said 'hi'

is false, because 'that buyer' refers to the same agent as the one that said 'hello' (which is Alice). This second condition would only be true if the messages ('say' 'hello') and ('say' ,'hi') had been sent by the same agent. Likewise, we have included the identifier 'he', which refers to the same agent as the last identifier that appeared earlier in the sentence. For example:

If any buyer has said 'hello' and he has said 'hi'

We may not want the rights of an agent to depend only on past events, but also on values of variables. Variables in SIMPLE are called **properties**. A property can be assigned to the protocol, or can be assigned to individual agents. For example, an auction may have a property 'highest bid' and each buyer may have a property 'bid price' to represent the price he or she has bid. If we have for example properties 'the price' and 'the account balance' then we can say things like:

If the price is lower than the account balance then any buyer can say 'buy'.
If the price is higher than 10 then the auctioneer can say 'sold'.

Properties can be added to a protocol by including property initialization sentences.

Definition 7. *A **property initialization sentence** is a sentence of one of the following forms:*

- *Initially,* `x` *is* `v`.
- *Every* `r` *has a* `x`*, which is initially* `v`.
- *Every* `r` *has an* `x`*, which is initially* `v`.

where `x` *can be any character string,* `v` *can be any character string, number, or identifier and* `r` *can be any singular role name.*

For example:

Every buyer has an age, which is initially 0.

Definition 8. *A **property condition** is a clause of one of the following forms:*

- `x` *is* `v`
- `x` *is not* `v`
- `x` *is higher than* `n`
- `x` *is lower than* `n`

where `x` *can be any character string,* `v` *can be any string, number or identifier, and* `n` *can be any number. The string* `x` *is called the **property name**, and* `v` *and* `n` *are called the **value**.*

Note that the current version of SIMPLE supports three types of properties: strings, numbers and identifiers. The type of a property is determined implicitly. That is: if the parser of the protocol is able to interpret the initial value of a property as a number, then the property is considered to be of type number, and likewise for identifiers. In all other cases the property is considered a string.

Definition 9. *A **property-update consequence** is a clause of the form:*

- `x` *becomes* `y`
- `x` *is* `v`
- `x` *is increased by* `n`
- `x` *is decreased by* `n`

where `x` *and* `y` *can be any character string,* `v` *can be any string, number of identifier, and* `n` *can be any number.*

Definition 10. *A **current-event condition** is a string of characters of one of the following forms:*

- `id` *says* '`x`'
- `id` *tells* `x`

where `id` *can be any identifier and* `x` *can be any character string.*

In order to change the values of properties (either assigned to the protocol or to an individual agent) we can use property-update rules.

Definition 11. *A **property-update rule** is a sentence of the form:*

- *When* `x` *then* `z`.

Where `x` *is a current-event condition and* `z` *is a conjunction of property-update consequences.*

Examples of property-update rules are:

When any buyer says 'bid!' then his bid price is increased by 10.
When the auctioneer says 'sold' then the last bidder becomes the winner.

Note that the clause *x becomes y* means that the value of property `y` is overwritten with the value of property `x`. This can be understood as follows: suppose we have a property called *Carol's sister* and a property called *Bob's wife*. Furthermore, suppose that *Carol's sister* is initialized to the value *'Alice'*. Then the clause *Carol's sister becomes bob's wife* means that the value *'Alice'* is copied into the property *Bob's wife*. Note that when a property is assigned to an agent we use the key word 'his' to refer to the agent that owns the property. To be precise: it refers to the last agent that appears earlier in the sentence. So in the above example, 'his bid price' refers to the property named 'bid price' assigned to the agent that said 'bid!'.

Another way that values of properties are updated is when a message of type ('tell', `x`, `y`) is sent. In that case the value `y` is assigned to a property with name `x`. For example, whenever an agent sends the message ('tell', 'the price', 100), the value 100 is automatically assigned to a property with the name 'the price'. The protocol does not need to contain any property-initialization sentence for such a property.

Definition 12. *A **right-update rule** is a sentence of the form:*

- `id` *can always say* `v`.
- `id` *can always tell* `v`.
- *If* `x` *then* `y`.
- *If* `x` *then* `y`*, as long as* `w`.

where `id` *is an identifier,* `v` *can be any character string,* `x` *and* `w` *are conjunctions of past-event conditions and/or property conditions and* `y` *is a conjunction of right-update consequences (the conditions in* `w` *are also referred to as **constraints**).*

Note that we allow such a rule to have no conditions at all, so that it is always active. In that case the protocol designer needs to include the keyword 'always' after the keyword 'can'. Also note that right-update rules are written in past tense, while property-update rules are written in present tense. This is because they are interpreted in a fundamentally different way, which we will explain in Section 5. Furthermore, we see in this definition that right-update rules may contain so-called *constraints*. A constraint is similar to a property condition, but is written at the end of the sentence, and indicated by the keywords *as long as*.

> *If the auctioneer has said 'open' then any buyer can tell his bid price, as long as his bid price is higher than the current price.*

A rule containing constraints is considered active if and only if all its conditions and constraints are satisfied. The difference between constraints and conditions, is that constraints refer to property values inside the consequences, whereas other conditions may only refer to past events or properties that do not appear inside the consequences. This distinction means that the truth of a condition is independent of any message, and therefore can already be determined before a specific message is sent, while the truth value of a constraint on a message X can only be determined after the participant has submitted message X, when the interpreter is verifying whether message X is legal. In the example sentence above for instance, the constraint says that the bid price told by the buyer, must be higher than the current price. This can of course only be checked *when* the buyer is telling his bid price, and not before.

Furthermore, we would like to remind the reader that right-update consequences can only have positive identifiers. This is important, because it means that a consequence can only *give* rights to an agent, but not take them away. Nevertheless, we can still make agents lose rights, but we do that by using negative conditions, rather than negative consequences. Take for example the following rule:

> *If the auctioneer has not said 'sold!' then any buyer can say 'bid!'.*

Here, every buyer initially has the right to say 'bid!', but loses that right once the auctioneer says 'sold!', because the condition becomes false (assuming there is no other active rule that gives the buyer the right to say 'bid!'). The big advantage of only allowing positive consequences, is that this makes it impossible to write inconsistent rules. An inconsistency would mean that there is one rule that specifies that you can do something, while another rule says you cannot do that. This is serious problem that one often encounters, for example in law. However, since we only allow positive consequences, this could never happen in our language.

Lemma 1. *A protocol written in SIMPLE is guaranteed to be free of inconsistencies.*

Proof. The proof is easy: in our language an agent has the right to do something if and only if there is an active rule with a consequence that gives this right to the agent. This can never lead to inconsistencies: either such a rule exists or not.

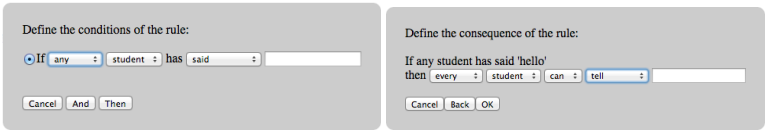

Fig. 1. Two screen shots of the SIMPLE editor. Users write sentences simply by selecting available options, and they can only write free text whenever the syntax rules indeed allow that. Therefore it is impossible to write malformed sentences.

5 The SIMPLE Interpreter

We will now describe the software component that interprets and enforces the protocols.

Whenever an agent tries to send a message, this message is first analyzed by the interpreter. The interpreter verifies if the agent sending the message indeed has the right to say that message and, if so, updates its internal state and forwards the message to the other agents connected to the server. If the sender of the message does not have the right to send that message he or she is notified that the message has failed. The message will in that case not be forwarded to the other agents and the internal state of the interpreter is not updated. In fact, we consider this message as not sent.

The internal state of the interpreter is defined as a list of all messages that have so far been sent successfully (the *event history*), a table that maps the name of each property to the current value of that property, a table that maps the name of each agent in the MAS to the role it is playing, and a table that maps the name of each agent in the MAS to a list of rights for that agent. Every time an agent tries to send a message, the interpreter follows the following procedure:

1. The list of rights of that agent is made empty.
2. For each right-update rule in the protocol, the interpreter verifies if its conditions are true:
   - If the condition is a property condition then it checks whether that property currently has the proper value to make the condition true.
   - If the condition is a past-event condition, the interpreter tries to find an event in the event history that matches the condition. If such an event is indeed found, then the condition is considered true.

   A rule for which all conditions are true is labeled as 'active'.
3. For each right-update consequence in each active rule, the interpreter checks whether the identifier matches the sender of the message and, if yes, adds the right corresponding to this consequence to the sender's list of rights. If this consequence has any constraints assigned to it, they are stored together with the right.
4. After all the rights of the sending agent have been determined the interpreter verifies whether any of them matches the message that the agent is trying to send.
5. Next, if the agent indeed has that right the interpreter checks whether its constraints (if any) are satisfied.
6. If the sending agent has the proper right, and all its constraints are satisfied then the interpreter determines if there are any property-update rules in the protocol

for which the condition matches the message. If yes, the properties in the rule's consequences are updated accordingly.
7. Finally, if the agent has the right to send the message and its constraints are satisfied, a copy of the message is stored in the event history, together with the name of the sender, and the message is forwarded to all other agents in the MAS.

It is important to note here that property-update rules and right-update rules are treated in a different way. To be precise: to verify whether a past-event condition is true, the interpreter compares the condition with all messages in the event history. Since messages are never removed from the event history this means that whenever a past-event condition becomes true, it remains true forever. For example, when a buyer says 'hello' then the condition *any buyer has said 'hello'* becomes true, and remains true forever. For negative conditions exactly the opposite holds: the condition *no buyer has said 'bye'* is initially true, but as soon as a buyer says 'bye' it becomes false, and will stay false forever.

The current-event conditions on the other hand are only considered true at the moment that the corresponding message is under evaluation of the interpreter. That is, the condition *when a buyer says hello* is considered to be true only while the interpreter is evaluating the message ('say', 'hello') sent by some agent playing the role of buyer. As soon as the interpreter handles the next message this condition is considered false again. The reason for this is that we consider that when you obtain a right, you keep that right for an extended period of time, until one of the negative conditions in the rule becomes false. Updating of a property on the other hand, is a one-time event that only takes place at the moment a certain message is sent.

6 Examples

We here provide two examples of protocols. Both have been tested and are correctly executed by the interpreter.

English Auction Protocol:

This protocol defines the role buyer (plural:buyers).
This protocol defines the role auctioneer (plural:auctioneers).
There must be exactly 1 auctioneer.
There must be at least 2 buyers.

Initially, the highest bidder is no one.
Initially, the winner is no one.
Initially, the current price is 0.
Every buyer has a bid price which is initially 0.

If the auctioneer has not said 'close' then he can say 'open'.
If the auctioneer has said 'open' then the auctioneer can say 'close'.
If the auctioneer has said 'open' and the auctioneer has not said 'close' then any buyer can tell his bid price, as long as his bid price is higher than the current price.

When a buyer tells his bid price then his bid price becomes the current price and he becomes the highest bidder.
When the auctioneer says 'close' then the highest bidder becomes the winner.

Dutch Auction Protocol:

This protocol defines the role buyer (plural:buyers).
This protocol defines the role auctioneer (plural:auctioneers).
There must be exactly 1 auctioneer.
There must be at least 2 buyers.

Initially, the price is 1000.
Initially, the winner is no one.

If no buyer has said 'mine' then the auctioneer can tell the next price, as long as the next price is lower than the price.
When the auctioneer tells the next price then the next price becomes the price.
If the auctioneer has told the price and no buyer has said 'mine' then any buyer can say 'mine'.
When a buyer says 'mine' then he becomes the winner.

7 Future Work

We consider that the language as it is, is still too limited to be of real practical use. We here list the shortcoming that we consider most important and that we plan to fix in the near future, as well as other improvements that we are considering.

Firstly, we will add the possibility to specify the recipient of a message. Currently every message is sent to all other agents in the MAS, which makes it impossible to send confidential information. This means we will allow to write sentences such as:

If the auctioneer has said 'welcome' to a buyer then that buyer can say 'hello' to the auctioneer.

Secondly, we would like the protocol designer to be able to express that a certain event must have taken place a certain number of times. For example:

If a buyer has told his bid price more than 5 times...

Thirdly, we would like to add more types of messages. and maybe even allow the protocol designer to define message types. That would make it possible to use certain domain-specific verbs. We could even take this a step further and allow the protocol designer to define new data types. Defining new types of objects is typically something that Inform 7 can handle well, so we may draw some inspiration from that language. Furthermore, we will add a system that determines at run time, whenever an agent tries to send an illegal message, which conditions first need to be fulfilled before the agent can indeed legally send that message. In this way the system can explain to the user why he or she made a mistake and will help the user to understand new protocols. In order to make the language more flexible and expressive, we will delve into literature about linguistics and apply some of its principles to our language. Finally, we are also considering the possibility to add support for model checking to our framework.

8 Acknowledgments

Supported by the Agreement Technologies CONSOLIDER project, contract CSD2007-0022 and INGENIO 2010 and CHIST-ERA project ACE and EU project 318770 PRAISE.

References

1. Alberti, M., Gavanelli, M., Lamma, E., Chesani, F., Mello, P., Torroni, P.: Compliance verification of agent interaction: a logic-based software tool. Applied Artificial Intelligence 20(2-4), 133–157 (2006)
2. Argente, E., Criado, N., Botti, V., Julian, V.: Norms for agent service controlling. EUMAS-08 pp. 1–15 (2008)
3. Artikis, A., Kamara, L., Pitt, J., Sergot, M.: A protocol for resource sharing in norm-governed ad hoc networks. In: Leite, J.a., Omicini, A., Torroni, P., Yolum, p. (eds.) Declarative Agent Languages and Technologies II, Lecture Notes in Computer Science, vol. 3476, pp. 221–238. Springer Berlin Heidelberg (2005), http://dx.doi.org/10.1007/11493402_13
4. Belnap, N., Perloff, M.: Seeing to it that: A canonical form for agentives. In: Kyburg, Henry E., J., Loui, R.P., Carlson, G.N. (eds.) Knowledge Representation and Defeasible Reasoning, Studies in Cognitive Systems, vol. 5, pp. 167–190. Springer Netherlands (1990), http://dx.doi.org/10.1007/978-94-009-0553-5_7
5. Broersen, J., Dignum, F., Dignum, V., Meyer, J.J.C.: Designing a deontic logic of deadlines. In: Deontic Logic in Computer Science, pp. 43–56. Springer Berlin Heidelberg (2004)
6. Cardoso, H.L., Urbano, J., Rocha, A.P., Castro, A.J., Oliveira, E.: Ante: Agreement negotiation in normative and trust-enabled environments. In: Ossowski, S. (ed.) Agreement Technologies, Law, Governance and Technology Series, vol. 8, pp. 549–564. Springer Netherlands (2013), http://dx.doi.org/10.1007/978-94-007-5583-3_32
7. Cranefield, S.: A rule language for modelling and monitoring social expectations in multi-agent systems. Tech. rep., In: Selected and Revised papers from the AAMAS 2005 Workshop on Agents, Norms and Institutions for Regulated Multiagent Systems. Volume 3913 of Lecture (2005)
8. Cranefield, S., Winikoff, M.: Verifying social expectations by model checking truncated paths. Journal of Logic and Computation 21(6), 1217–1256 (2011), http://logcom.oxfordjournals.org/content/21/6/1217.abstract
9. Dastani, M., Tinnemeier, N.A., Meyer, J.J.C.: A programming language for normative multi-agent systems (2009)
10. Esteva, M., de la Cruz, D., Sierra, C.: Islander: en electronic institutions editor. vol. 3, pp. 1045–1052. ACM PRESS, Bologna, Italy (July 15-19 2002)
11. Esteva, M., Rodríguez-Aguilar, J.A., Arcos, J.L., Sierra, C., Noriega, P., Rosell, B., de la Cruz, D.: Electronic institutions development environment. In: AAMAS (Demos). pp. 1657–1658 (2008), http://www.iiia.csic.es/files/pdfs/eide.pdf
12. Fornara, N., Cardoso, H.L., Noriega, P., Oliveira, E., Tampitsikas, C., Schumacher, M.I.: Modelling Agent Institutions, chap. 18, pp. 277–307. No. 8, Springer-Verlag GmdH (2013)
13. García-Camino, A.: Ignoring, forcing and expecting simultaneous events in electronic institutions. In: Proceedings of the 2007 International Conference on Coordination, Organizations, Institutions, and Norms in Agent Systems III. pp. 15–26. COIN'07, Springer-Verlag, Berlin, Heidelberg (2008), http://dl.acm.org/citation.cfm?id=1791649.1791652
14. Governatori, G., Rotolo, A., Sartor, G.: Temporalised normative positions in defeasible logic. In: Procedings of the 10th International Conference on Artificial Intelligence and Law. pp. 25–34. ACM Press (2005)

15. van der Hoek, W., Roberts, M., Wooldridge, M.: Social laws in alternating time: effectiveness, feasibility, and synthesis. Synthese 156(1), 1–19 (2007), http://dx.doi.org/10.1007/s11229-006-9072-6
16. Hübner, J.F., Sichman, J.S.a., Boissier, O.: S-moise+: A middleware for developing organised multi-agent systems. In: Boissier, O., Padget, J., Dignum, V., Lindemann, G., Matson, E., Ossowski, S., Sichman, J.S.a., Vázquez-Salceda, J. (eds.) Coordination, Organizations, Institutions, and Norms in Multi-Agent Systems, Lecture Notes in Computer Science, vol. 3913, pp. 64–77. Springer Berlin Heidelberg (2006), http://dx.doi.org/10.1007/11775331_5
17. de Jonge, D., Rosell, B., Sierra, C.: Human interactions in electronic institutions. In: Chesnevar, C., Onaindia de la Rivaherrera, E., Ossowski, S., Vouros, G. (eds.) 2nd International Conference on Agreement Technologies. vol. 8068, pp. 75–89. Springer, Springer, Beijing, China (01/08/2013 2013)
18. Kollingbaum, M.J.: Norm-governed practical reasoning agents. Ph.D. thesis, University of Aberdeen (2005)
19. Kröger, F.: Temporal Logic of Programs. Springer-Verlag New York, Inc., New York, NY, USA (1987)
20. Lewis, D.: Semantic analyses for dyadic deontic logic. In: Stenlund, S. (ed.) Logical Theory and Semantic Analysis: Essays Dedicated to Stig Kanger on His Fiftieth Birthday, pp. 1–14. Reidel, Dordrecht (1974)
21. Lopez y Lopez, F., Luck, M., Puebla, A.: A model of normative multi-agent systems and dynamic relationships. In: Regulated Agent-Based Social Systems. Volume 2934 of Lecture Notes in Artificial Intelligence. pp. 259–280. Springer (2004)
22. Makinson, D., Van Der Torre, L.: Input/output logics. Journal of Philosophical Logic 29(4), 383–408 (2000)
23. Meyer, J.J.C.: A different approach to deontic logic: deontic logic viewed as a variant of dynamic logic. Notre Dame J. Formal Logic 29(1), 109–136 (12 1987), http://dx.doi.org/10.1305/ndjfl/1093637776
24. Nelson, G.: Natural language, semantic analysis and interactive fiction. http://inform7.com/learn/documents/WhitePaper.pdf (2014)
25. Nute, D.: Defeasible Deontic Logic. Springer (1997)
26. Sergot, M., Craven, R.: The deontic component of action language nc+. In: Goble, L., Meyer, J.J.C. (eds.) Deontic Logic and Artificial Normative Systems, Lecture Notes in Computer Science, vol. 4048, pp. 222–237. Springer Berlin Heidelberg (2006), http://dx.doi.org/10.1007/11786849_19
27. Tampitsikas, C., Bromuri, S., Schumacher, M.: Manet: A model for first-class electronic institutions. In: Cranefield, S., Vazquez-Salceda, J., van Riemsdijk, B., Noriega, P. (eds.) Coordination, Organizations, Institutions, and Norms in Agent Systems VII. Lecture Notes in Artificial Intelligence, vol. 7254. Springer Verlag (2012), http://link.springer.com/chapter/10.1007/978-3-642-35545-5_5
28. 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. Policies for Distributed Systems and Networks, IEEE International Workshop on 0, 145–152 (2008)
29. Vázquez-Salceda, J., Aldewereld, H., Dignum, F.: Implementing norms in multiagent systems. Multiagent system technologies pp. 313–327 (2004)
30. von Wright, G.H.: Deontic logic. Mind 60, 1–15 (1951)