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 have been designed to be used by computer scientists and are not suitable for people with no more than average computer skills. Since more and more tools are nowadays 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, as well as 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 origins of the agents are 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 [7], MANET [34], S-MOISE+ [22], and EIDE [15]. 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*.

V. Dignum et al. (Eds.): COIN 2015, LNAI 9628, pp. 98–118, 2016.
DOI: 10.1007/978-3-319-42691-4_6

One common characteristic of these systems is that they are mainly designed with computer scientists as their target users. They require knowledge of multi-agent 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 [23].

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 and makes sure that all sentences are syntactically correct.

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. Protocols written in this language do not specify what the agents *must* do, but only 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. We still consider the language as presented here (version 0.10) to be in a premature state, and we plan to extend it much more in the future. A working demonstration of the SIMPLE editor and interpreter can be found at http://simple.iiia.csic.es.

The rest of this paper is organized as follows: in Sect. 2 we give a short overview of previous work done in this field. Next, in Sect. 3 we explain the assumptions that we have made about the set-up of any MAS to which our language is applied. In Sect. 4 we describe the syntax rules of our language. Next, in Sect. 5 we explain how our interpreter parses text files written in our language and enforces its rules upon the agents. Then, in Sect. 6 we give two examples of protocols written in SIMPLE, for which we have tested that they are successfully parsed and enforced by our interpreter. In Sect. 7 we make a comparison between the expressivity of SIMPLE and the expressivity of the existing Islander tool. And finally, in Sect. 8 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 [7], MANET [34], S-MOISE+ [22], and EIDE [15]. A comparative study of some of those systems has been made in [16].

ANTE [7] 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 [34] 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* [14] 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) [37]. Important refinements of this logic are Dyadic Deontic Logic (DDL) [26] and Defeasible Deontic Logic [31]. Furthermore, an extension of this taking temporal considerations into account was proposed in [20]. In [28] a system to formalize norms using input/output logic was proposed, while in [21] the authors provide a model for the formalization of social law by means of Alternating-time Temporal Logic (ATL). In [25] the author proposes the use of Linear Time Logic (LTL) to express norms. Other important approaches are based on Propositional Dynamic Logic (PDL) [29], on See-to-it-that logic (STIT) [4] and on Computational Tree Logic (CTL) [6]. Models for the verification of expectations in normative systems are proposed in [1,10], and in [32] 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 [27] 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 [11]. Other important examples of languages and frameworks for the implementation of norms and rules are described in: [2,9,18,24,35,36].

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.

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 [30]. 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.

Another example of an easy-to-use language is If-This-Then-That[3] (IFTTT). This tool allows users to define if-then rules that trigger some action to occur whenever a certain event takes places. This concept is very similar to SIMPLE, except that in SIMPLE the rules do not trigger events to take place, but rather grant rights to agents.

Controlled natural language has been applied to policy making before in [8,12], which is essentially a mapping between Attempto Controlled English (ACE) [17] and the policy specification language Protune [5]. However, this work seems to focus mainly on the specification of static rules, whereas our work puts emphasis on dynamic rules that may change depending on events that are happening during the execution of the policy. This is reflected by the fact that in their language the conditions of the rules are written in simple present, rather than in present perfect as in our language. A similar tool to write static rules in controlled natural language was presented in [33].

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 intended 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] http://en.wikipedia.org/wiki/HyperTalk.
[2] http://www.osmosian.com.
[3] https://ifttt.com/.

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 contains the protocol as a set of sentences that follow the SIMPLE syntax, and are therefore human readable. Furthermore, it also stores the protocol in JSON format so that it can be parsed easily by the interpreter.

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`)
– ('announce', `y`, `z`)

in which the sender can replace `x`, `y` and `z` by any character string (we will see later that the 'announce' message has the interpretation that, by uttering this message, the value of `z` will be assigned to the variable `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`)
– ('announce', `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 keyword 'anything'. A right ('announce', `w`) matches a message ('announce', `y`, `z`) if and only if `y` equals `w`. For example: if the agent has the right ('announce', 'price') then it matches the message ('announce', '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.

4.1 Roles

In order to define a role in the protocol the user must first specify two names for that role: the **singular role name** and the **plural role name**, for example: 'auctioneer' and 'auctioneers'. The user must then specify a role constraint sentence:

Definition 1. *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 the plural role name, except in the case that* `x` $= 1$ *in which case* `r` *it is the singular role name.*

The following sentence is an example 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 on the other hand there are not enough participants for every role, then every message is considered illegal. Therefore, the agents cannot start sending messages to one another until there are enough participants for every role.

4.2 Conditions and Consequences

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 announce 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 2. *A* ***quantifier*** *is any of these keywords: no, any, every, a, an, the, that.*

Definition 3. *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 4. *A* ***past-event condition*** *is a string of characters of one of the following forms:*

– `id` *has said '*`x`*'*
– `id` *has announced the* `x`
– `id` *has announced his* `x`
– `pid` *has not said '*`x`*'*
– `pid` *has not announced the* `x`
– `pid` *has not announced his* `x`

where `id` *can be any identifier,* `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 condition is a specific type of condition. We will define other types of condition later on. A positive past-event condition is considered true if and only 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 and only if there is no message in the event history that matches the condition.

Definition 5. *A* ***right-update consequence*** *is a string of characters of one of the following forms:*

- `pid` *can say '*`x`*'*
- `pid` *can announce the* `x`
- `pid` *can announce his* `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 right-update consequence. 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:

a buyer has said 'hello' and a buyer has said 'hi'

is true. However, the condition:

a 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 a buyer has said 'hello' and he has said 'hi'

4.3 Properties

The rights of an agent may not only depend on past events, but may also depend on values of variables. Variables in SIMPLE are called **properties**. A property can be assigned to the protocol (a *global property*), or can be assigned to individual agents (a *role property*). For example, we may specify that every buyer has a property 'age', and that the protocol has a global property 'minimum age', so that we can state conditions such as:

If a buyer has said 'hello' and his age is greater than the minimum age then...

A property can be defined by including a *property initialization sentence* in the protocol.

Definition 6. *A **property initialization sentence** is a sentence of one of the following forms:*

- *This protocol has a* `x`*, which is initially* `v`*.*
- *Every* `r` *has a* `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. The string* `x` *is called the **property name**, and* `v` *is its **initial value**.*

The first of these sentences is used to define a global property, while the second one defines a role property. If the name of the property `x` starts with a vowel then the editor will automatically replace the article 'a' in the sentence with 'an'. For example:

Every buyer has an age, which is initially 0.

A property can also be added to a protocol without including a property initialization sentence, but instead by mentioning it in some rule containing the verb 'to announce'. For example, if there is a rule containing the condition

If a buyer has announced his age...

then the interpreter automatically understands that the role 'buyer' has a property named 'age'. Similarly, if the protocol contains a sentence containing the conditions

If the auctioneer has announced the start price...

then the interpreter understands that the protocol has a global property named 'start price'.

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

- x *is less than* n
- x *is less than or equal to* n
- x *is* v
- x *is not* v
- x *is greater than or equal to* n
- x *is greater than* n

where x *is either the keyword 'the' followed by the name of a global property, or the keyword 'his' followed by the name of a role property.* v *can be any string, number or identifier, and* n *can be any number.*

Definition 8. *A **property-update consequence** is a clause of the form:*

- x *becomes* y
- x *will be* v
- x *is increased by* n
- x *is decreased by* n
- x *is multiplied by* n
- x *is divided by* n

where x *and* y *both are either the keyword 'the' followed by the name of a global property, or the keyword 'his' followed by the name of a role property.* y *can be any character string,* v *can be any string, number of identifier, and* n *can be any number.*

Definition 9. *A **current-event condition** is a string of characters of one of the following forms:*

- pid *says '*x*'*
- pid *announces the* x
- pid *announces his* x

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

In order to change the values of properties we can use property-update rules.

Definition 10. *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 property-update consequence.*

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 ('announce', `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 ('announce', 'price', 100), the value 100 is automatically assigned to a property with the name 'price'. More specifically, if the property 'price' is global than that unique property is updated, while if it is a role property, for example for the role 'buyer', and the sender of the message indeed plays that role, then it is the property of the sender that is updated. If neither is the case, that is: if the property 'price' is a role property for the role 'buyer', but the sender does not play the 'buyer' role, then the message is illegal.

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

- `id` *can always say* `v`*.*
- `id` *can always announce the* `v`*.*
- `id` *can always announce his* `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 right-update consequence (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 have past-event conditions (which are written in present perfect), while property-update rules have current-event conditions (which are written in simple present). This is because they are interpreted in a fundamentally different way, which we will explain in Sect. 5.

4.4 Constraints

We have seen in Definition 11 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 announce his bid price, as long as his bid price is higher than the current price.

The consequences of a rule only have effect if all conditions and constraints of the rule are satisfied. The difference between constraints and conditions is that constraints refer to property values inside the consequence of the sentence, whereas conditions may only refer to past events or properties that do not appear inside the consequence. This means that when the interpreter verifies the legality of a certain message X, the truth of the constraints of any rule depend on the contents of that message, whereas the truth of the conditions of any rule can already be determined before the interpreter has received message X.

In the example sentence above for instance, the constraint says that the bid price announced by the buyer, must be higher than the current price. This can of course only be checked *when* the buyer is announcing his bid price, and not before.

4.5 Inconsistencies

One very important aspect of our language is that right-update consequences can only have *positive* identifiers. This 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!'.

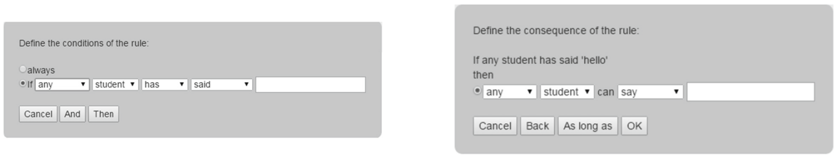
Here, every buyer initially has the right to say 'bid!'. If there is no other rule that gives buyers the right to do that, then buyers will lose this right once the auctioneer says 'sold!', because the condition becomes false. If there is more than one rule that grants the right to say 'bid!' to every buyer then all those rules must become inactive in order for the buyers to lose that rule.

The big advantage of only allowing positive consequences, is that this makes it impossible to write inconsistent rules. Recall from Sect. 3 that for every message submitted the interpreter needs to answer the question: "Does the sender of this message have the right to do so?", with either "yes" or "no". We say that a protocol is *consistent* if for every possible message this question has only one correct answer.

Lemma 1. *A protocol written in SIMPLE is guaranteed to be consistent.*

Proof. The proof is easy: in our language, by definition, an agent has the right to do something if and only if there is at least one active rule that grants this right to the agent. This can never lead to inconsistencies: either such a rule exists or not.

This aspect certainly does not make our language unique, as the same principle applies to several other logical languages, such as GDL [19] and ASP [13] (Fig. 1).


Define the conditions of the rule:

always
If any student has said

Define the consequence of the rule:

If any student has said 'hello'
then
any student can say

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 send 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 consists of the following data structures:

– 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
– 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:[4]

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'.

[4] This procedure can be implemented in a much more efficient way than presented here, but we think this is not very relevant for this paper, so we prefer to present it in a way that is easier to understand for the reader.

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:

There must be exactly 1 auctioneer.
There must be at least 2 buyers.

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

If the auctioneer has not said 'sold!' and the auctioneer has not announced the current price, then the auctioneer can announce the current price.
If the auctioneer has not said 'sold!' and the auctioneer has announced the current price, then any buyer can announce his bid price, as long as bid price is greater than current price.
When a buyer announces his bid price, then his bid price becomes the current price.
When a buyer announces his bid price, then that buyer becomes the highest bidder.
If any buyer has announced his bid price, then the auctioneer can say 'sold!'.
When the auctioneer says 'sold!', then the highest bidder becomes the winner.

Dutch Auction Protocol:

There must be exactly 1 auctioneer.
There must be at least 2 buyers.

This protocol has a current price, which is initially 0.
This protocol has a winner, which is initially no one.

If the auctioneer has not announced the current price, then he can announce the current price.
If the auctioneer has announced the current price and no buyer has said 'mine!', then the auctioneer can say 'next!'.
When the auctioneer says 'next!', then the current price is decreased by 1.
If the auctioneer has announced the current price and no buyer has said 'mine!', then any buyer can say 'mine!'.
When a buyer says 'mine!', then that buyer becomes the winner.

7 Comparison with Islander

Many other tools have been developed for the specification of protocols so it would be impossible to discuss all the advantages and disadvantages of SIMPLE with respect to those existing tools. Therefore, we limit ourselves to a comparison with the Islander tool, which is part of the EIDE framework.

Islander allows users to specify protocols using a graphical model that represents protocols as a directed graph in which the nodes represent states, and edges between nodes represent actions (an agent sending a message to another agent) that lead from one state to the next. This graphical model can be extended by assigning pre-conditions and post-conditions to the edges which are written in a formal language.

The advantage of the graphical model is that is relatively easy to use and understand, but the disadvantage is that it has very limited expressivity. Therefore, it is almost always necessary to use it in combination with the formal language, which however can be very difficult to use, even for computer scientists.

We will now give an example of a protocol that is hard to express using only the graphical representation of Islander. Note that this example uses the verb 'to make' which is currently not yet available in SIMPLE. Therefore, the idea is to show how SIMPLE *will* be more easy to use than Islander in the future, when we have further extended the language.

Suppose we want to implement the following protocol in Islander:

There must be at least 5 students.
Initially, any student can make assignment 1.
If a student has made assignment 1 then he can make assignment 2.
If a student has made assignment 2 then he can make assignment 3.

If there would be only 1 student then this protocol would be very easy to implement in Islander. It would just be a linear graph with 4 nodes and three edges, where each edge corresponds to making an assignment.

With multiple students however, you run into the problem that students may make their assignments at different speeds. For example, one student may quickly deliver assignments 1 and 2, while another is still busy with assignment 1. This means that at any moment each student can be in any of 4 states, and thus the protocol as a whole can be in any of 4^n different states if there are n students. With as little as 5 students drawing the graph would already become practically impossible as it would require the designer to draw $4^5 = 1024$ states plus all their edges. The only realistic way to specify this protocol would be to use the formal language rather than the graph-representation.

Another way to implement this protocol in Islander would be to implement it as a protocol for only 1 student, and then let n of these protocols run in parallel. In this way we again only need to draw a linear graph with 4 nodes and 3 edges. However, this is only possible because in this example there is no interdependency between the students' actions. If we make the example a bit more complicated, for example by adding a final exam that only starts when all students have finished all assignments, this is no longer possible.

Another problem with Islander is that one cannot use universal quantifiers. Even in Islander's formal language one cannot directly state something like *"If all students have finished their assignments..."*. The only way to achieve this is to create a list of names of students, make sure that the name of a student is added to this list when he or she enters the institution, and make sure that the name of a student is removed from this list whenever he or she finishes his or her assignment. Then, whenever you need the precondition that all students must have finished their assignments, you can specify that this list must be empty. In practice, this turns out to be a tedious job to do all this in Islander, making it far from user-friendly.

8 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 announced his bid price more than 5 times...*

Thirdly, we would like to add time-constraints to the language, so that we could define rights that expire after a certain amount of time, such as:

> *If no one announces his bid during* 10 s *then the highest bidder becomes the winner.*

Furthermore, 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. For example:

> *If a student has finished his assignment...*

We could even take this a step further and allow the protocol designer to define new data types, similar to data types in the EIDE framework. For example, one could define a data type "contract" by including a sentences such as:

> *A contract consists of a date, a price, and a quantity.*
> *A price is a positive number.*

One could then define a negotiation protocol with sentences such as

> *Any negotiator can propose a contract.*

If such data types are composed of basic types such as Strings and numbers then during the execution of the protocol the GUI can display the proper input fields for the user to specify the details of the contract to propose. 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 will perform an empirical study to evaluate how easy-to-use this language really is. We will let random people implement a protocol using our language as well as using some other existing tool such as Islander to compare whether our language indeed makes the task easier.

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. Appl. Artif. Intell. **20**(2–4), 133–157 (2006)
2. Argente, E., Criado, N., Botti, V., Julian, V.: Norms for agent service controlling. In: 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., Omicini, A., Torroni, P., Yolum, P. (eds.) DALT 2004. LNCS (LNAI), vol. 3476, pp. 221–238. Springer, 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 Jr., H.E., 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. Bonatti, P.A., Olmedilla, D.: Driving and monitoring provisional trust negotiation with metapolicies. In: 6th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2005), 6–8 June 2005, Stockholm, Sweden, pp. 14–23 (2005). http://dx.doi.org/10.1109/POLICY.2005.13
6. Broersen, J., Dignum, F., Dignum, V., Meyer, J.-J.C.: Designing a deontic logic of deadlines. In: Lomuscio, A., Nute, D. (eds.) DEON 2004. LNCS (LNAI), vol. 3065, pp. 43–56. Springer, Heidelberg (2004)
7. 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
8. Coi, J.L.D., Kärger, P., Olmedilla, D., Zerr, S.: Using natural language policies for privacy control in social platforms (2009). http://CEUR-WS.org/Vol-447/paper4.pdf

9. Cranefield, S.: A rule language for modelling and monitoring social expectations in multi-agent systems. In: Boissier, O., Padget, J., Dignum, V., Lindemann, G., Matson, E., Ossowski, S., Sichman, J.S., Vázquez-Salceda, J. (eds.) ANIREM and OOOP 2005. LNCS (LNAI), vol. 3913, pp. 246–258. Springer, Heidelberg (2006)
10. Cranefield, S., Winikoff, M.: Verifying social expectations by model checking truncated paths. J. Logic Comput. **21**(6), 1217–1256 (2011). http://logcom.oxfordjournals.org/content/21/6/1217.abstract
11. Dastani, M., Tinnemeier, N.A., Meyer, J.J.C.: A programming language for normative multi-agent systems (2009)
12. De Coi, J.: Notes for a possible ACE → Protune mapping. Technical report, Forschungszentrum L3S, Appelstr. 9a, 30167 Hannover, July 2008
13. Eiter, T., Ianni, G., Krennwallner, T.: Answer set programming: a primer. In: Tessaris, S., Franconi, E., Eiter, T., Gutierrez, C., Handschuh, S., Rousset, M.-C., Schmidt, R.A. (eds.) Reasoning Web. LNCS, vol. 5689, pp. 40–110. Springer, Heidelberg (2009). http://dx.doi.org/10.1007/978-3-642-03754-2_2
14. Esteva, M., de la Cruz, D., Sierra, C.: Islander: en electronic institutions editor. In: Bologna, Italy, vol. 3, pp. 1045–1052. ACM Press, 15–19 July 2002
15. 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
16. Fornara, N., Cardoso, H.L., Noriega, P., Oliveira, E., Tampitsikas, C., Schumacher, M.I.: Modelling agent institutions. In: Ossowski, S. (ed.) Agreement Technologies, Chap. 18, vol. 8, pp. 277–307. Springer-Verlag GmdH, Netherlands (2013)
17. Fuchs, N.E., Kaljurand, K., Kuhn, T.: Attempto controlled English for knowledge representation. In: Baroglio, C., Bonatti, P.A., Małuszyński, J., Marchiori, M., Polleres, A., Schaffert, S. (eds.) Reasoning Web. LNCS, vol. 5224, pp. 104–124. Springer, Heidelberg (2008). http://dx.doi.org/10.1007/978-3-540-85658-0_3
18. García-Camino, A.: Ignoring, forcing and expecting simultaneous events in electronic institutions. In: Sichman, J.S., Padget, J., Ossowski, S., Noriega, P. (eds.) COIN 2007. LNCS (LNAI), vol. 4870, pp. 15–26. Springer, Heidelberg (2008). http://dl.acm.org/citation.cfm?id=1791649.1791652
19. Genesereth, M., Love, N., Pell, B.: General game playing: overview of the aaai competition. AI Mag. **26**(2), 62–72 (2005)
20. 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)
21. 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
22. Hübner, J.F., Sichman, J.S., 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., Vázquez-Salceda, J. (eds.) ANIREM and OOOP 2005. LNCS (LNAI), vol. 3913, pp. 64–78. Springer, Heidelberg (2006). http://dx.doi.org/10.1007/11775331_5
23. de Jonge, D., Rosell, B., Sierra, C.: Human interactions in electronic institutions. In: Chesñevar, C.I., Onaindia, E., Ossowski, S., Vouros, G. (eds.) AT 2013. LNCS, vol. 8068, pp. 75–89. Springer, Heidelberg (2013)
24. Kollingbaum, M.J.: Norm-governed practical reasoning agents. Ph.D. thesis, University of Aberdeen (2005)
25. Kröger, F.: Temporal Logic of Programs. Springer-Verlag New York, Inc., New York (1987)

26. 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)
27. López y López, F., Luck, M.: A model of normative multi-agent systems and dynamic relationships. In: Lindemann, G., Moldt, D., Paolucci, M. (eds.) RASTA 2002. LNCS (LNAI), vol. 2934, pp. 259–280. Springer, Heidelberg (2004)
28. Makinson, D., Van Der Torre, L.: Input/output logics. J. Philos. Logic **29**(4), 383–408 (2000)
29. 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 (1987). http://dx.doi.org/10.1305/ndjfl/1093637776
30. Nelson, G.: Natural language, semantic analysis and interactive fiction (2014). http://inform7.com/learn/documents/WhitePaper.pdf
31. Nute, D.: Defeasible Deontic Logic. Springer, The Netherlands (1997)
32. Sergot, M.J., Craven, R.: The Deontic Component of Action Language $nC+$. In: Goble, L., Meyer, J.-J.C. (eds.) DEON 2006. LNCS (LNAI), vol. 4048, pp. 222–237. Springer, Heidelberg (2006). http://dx.doi.org/10.1007/11786849_19
33. Shi, L.L., Chadwick, D.W.: A controlled natural language interface for authoring access control policies. In: Proceedings of the 2011 ACM Symposium on Applied Computing (SAC), TaiChung, Taiwan, 21–24 March 2011, pp. 1524–1530 (2011). http://doi.acm.org/10.1145/1982185.1982510
34. Tampitsikas, C., Bromuri, S., Schumacher, M.I.: MANET: a model for first-class electronic institutions. In: Cranefield, S., van Riemsdijk, M.B., Vázquez-Salceda, J., Noriega, P. (eds.) COIN 2011. LNCS, vol. 7254, pp. 75–92. Springer, Heidelberg (2012). http://link.springer.com/chapter/10.1007/978-3-642-35545-5_5
35. 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, pp. 145–152 (2008)
36. Vázquez-Salceda, J., Aldewereld, H., Dignum, F.: Implementing norms in multiagent systems. In: Lindemann, G., Denzinger, J., Timm, I.J., Unland, R. (eds.) MATES 2004. LNCS (LNAI), vol. 3187, pp. 313–327. Springer, Heidelberg (2004)
37. von Wright, G.H.: Deontic logic. Mind **60**, 1–15 (1951)