Model Checking Normative Agent Organisations*

Louise Dennis[1], Nick Tinnemeier[2], and John-Jules Meyer[2]

[1] Department of Computer Science, University of Liverpool, Liverpool, U.K.
L.A.Dennis@csc.liv.ac.uk
[2] Department of Information and Computing Sciences,
Utrecht University, Utrecht, The Netherlands
{nick,jj}@cs.uu.nl

Abstract. We present the integration of a normative programming language in the MCAPL framework for model checking multi-agent systems. The result is a framework facilitating the implementation and verification of multi-agent systems coordinated via a normative organisation. The organisation can be programmed in the normative language while the constituent agents may be implemented in a number of (BDI) agent programming languages.

We demonstrate how this framework can be used to check properties of the organisation and of the individual agents in an LTL based property specification language. We show that different properties may be checked depending on the information available to the model checker about the internal state of the agents. We discuss, in particular, an error we detected in the organisation code of our case study which was only highlighted by attempting a verification with "white box" agents.

1 Introduction

Since Yoav Shoham coined the term "agent-oriented programming" [19], many dedicated languages, interpreters and platforms to facilitate the construction of multi-agent systems have been proposed. Examples of such agent programming languages are Jason [6], GOAL [13] and 2APL [8]. An interesting feature of the agent paradigm is the possibility for building heterogeneous agent systems. That is to say, a system in which multiple agents, implemented in different agent programming languages and possibly by different parties, interact. Recently, the area of agent programming is shifting attention from constructs for implementing single agents, such as goals, beliefs and plans, to social constructs for programming multi-agent systems, such as roles and norms. In this view a multi-agent system is seen as a computational organisation that is constructed separately from the agents that will interact with it. Typically, little can be assumed about the internals of these agents and the behaviour they will exhibit. When little can be assumed about the agents that will interact with the organisation, a norm enforcement mechanism – a process that is responsible for detecting when norms are violated and responding to these violations by imposing sanctions – becomes crucial

* Work partially supported by EPSRC under grant EP/D052548 and by the CoCoMAS project funded through the Dutch Organization for Scientific Research (NWO).

to regulate their behaviour and to achieve and maintain the system's global design objectives [20].

One of the challenges in constructing multi-agent systems is to verify that the system meets its overall design objectives and satisfies some desirable properties. For example, that a set of norms actually enforces the intended behaviour and whether the agents that will reside in the system will be able to achieve their goals. In this paper we report on the extension of earlier work [11] of one of the authors on the automatic verification of heterogeneous agent systems to include organisational (mostly normative) aspects also, by incorporating the normative programming language as presented in [9]. The resulting framework allows us to use automated verification techniques for multi-agent systems consisting of a heterogeneous set of agents that interact with a norm governed organisation. The framework in [11] is primarily targeted at a rapid implementation of agent programming languages that are endowed with an *operational semantics* [16]. The choice for the integration of the normative programming language proposed in [9] is mainly motivated by the presence of an operational semantics which facilitates the integration with [11].

It should be noted that we are not the first to investigate the automatic verification of multi-agent systems and computational organisations. There are already some notable achievements in this direction. Examples of work on model checking techniques for multi-agent systems are [4,5,15]. In contrast to [11] the work on model checking agent systems is targeted at homogeneous systems pertaining to the less realistic case in which all agents are built in the same language. Most importantly, these works (including [11]) do not consider the verification of organisational concepts. Work related to the verification of organisational aspects has appeared, for example, in [14,7,21,1], but in these frameworks the internals of the agents are (intentionally) viewed as unknown. This is explained by the observation that in a *deployed* system little can be assumed about the agents that will interact with it. Still, we believe that for verification purposes at *design time* it would be useful to also take the agents' architecture into account. Doing so allows us, for example, to assert the correctness of a (prototype) agent implementation in the sense that it will achieve its goals without violating a norm. In designing a normative organisation a programmer puts norms into place to enforce desirable behaviour of the participating agents. Implementing prototypical agents and employing them in the organisation allows us to verify whether the actual behaviour accords with the intended behaviour of the system as a whole. A proven prototypical implementation of a norm-abiding agent might then be published to serve as a guideline for external agent developers.

The rest of the paper is structured as follows: In section 2 we give an overview of the language for programming normative organisations (which we will name ORWELL from now on) and discuss the general properties of the agent variant of the dining philosophers problem we use as a running example throughout the paper. Section 3 describes the MCAPL framework for model checking multi-agent systems programmed in a variety of BDI-style agent programming languages. Section 4 discusses the implementation of ORWELL in the MCAPL framework. Section 5 discusses a case study we undertook to model check some properties in a number of different multi-agent systems using the organisation.

2 ORWELL Programming Normative Agent Organisations

This section briefly explains the basic concepts involved in the approach to constructing normative multi-agent organisations and how they can be programmed in ORWELL. A more detailed description of its formal syntax and operational semantics together with an example involving a conference management system can be found in [9].

A multi-agent system, as we conceive it, consists of a set of heterogeneous agents interacting with a normative organisation (henceforth organisation). Figure 1 depicts a snapshot of such a multi-agent system. As mentioned before, by heterogeneous we mean that agents are potentially implemented in different agent programming languages by unknown programmers. An organisation encapsulates a domain specific state and function, for instance, a database in which papers and reviews are stored and accompanying functions to upload them. The domain specific state is modeled by a set of *brute facts*, taken from Searle [18]. The agents perform actions that change the brute state to interact with the organisation and exploit its functionality. The general structure of a multi-agent system we adopt is indeed inspired by the agents and artifacts approach of Ricci et al. [17] in which agents exploit artifacts to achieve their design objectives.

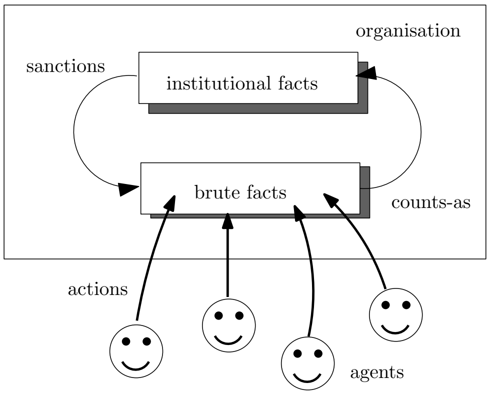


Fig. 1. Agents interacting with a normative organisation

An important purpose of an organisation is to coordinate the behavior of its interactants and to guide them in interacting with it in a meaningful way. This is achieved through normative component that is defined by a simple account of *counts-as rules* as defined by Grossi [12]. Counts-as rules normatively assess the brute facts and label a state with a normative judgment marking brute states as, for example, good or bad. An uploaded paper that exceeds the page limit would, for instance, be assessed as an undesirable state. The normative judgments about the brute state are stored as *institutional facts*, again taken from Searle [18]. To motivate the agents to abide by the norms, certain normative judgments might lead to sanctions which are imposed on the brute state, for example rejecting a paper that violates the page limit by removing it from the database.

In what follows we explain all these constructs using the agent variant of the famous dining philosophers problem in which five spaghetti-eating agents sit at a circular table and compete for five chopsticks. The sticks are placed in between the agents and each agent needs two sticks to eat. Each agent can only pickup the sticks on her immediate left and right. When not eating the agents are deliberating. It is important to emphasize that in this example the chopsticks are metaphors for shared resources and the problem touches upon many interesting problems that commonly arise in the field of concurrent computing, in particular deadlock and starvation. The problem in all its simplicity is, however, also interesting in the context of multi-agent systems (which are characterised by distribution and concurrency) in general, and organisation-oriented approaches in particular. Solutions of how the agents can efficiently share the resources can be considered protocols, which as will be shown naturally translate into norms. There are many known solutions to the dining philosophers problem and it is not our intention to come up with a novel solution. We merely use it to illustrate the ORWELL language.

The ORWELL implementation of the dining agents is listed in code fragment 2.1 (and continued in code fragment 2.2.) The initial brute state of the organisation is specified by the facts component. The agents named `ag1,...,ag5` are numbered one to five clockwise through facts of the form `agent(A,I)`. Sticks are also identified by a number such that the right stick of an agent numbered `I` is numbered `I` and its left stick is numbered `I%5+1`[1]. The fact that an agent `I` is holding a stick is modeled by `hold(I,X)` with `X` $\in$ `{r,l}` in which `r` denotes the right and `l` the left stick. The fact that a stick `I` is down on the table is denoted by `down(I)` and a fact `food(I)` denotes that there is food on the plate of agent `I`. We assume that initially no agent is holding a stick (all sticks are on the table) and all agents are served with food. The initial situation of the dining agents is shown graphically in figure 2. The specification of the initial brute state is depicted in lines 1-4.

The brute facts change under the performance of actions by agents. The effects describe how the brute state may evolve under the performance of actions. They are used by the organization to determine the resulting brute state after performance of the action. They are defined by triples of the form $\{Pre\}a\{Post\}$, intuitively meaning that when action a is executed and set of facts Pre is derivable by the current brute state, the set of facts denoted by $Post$ is to be accomodated in it. We use the notation ϕ to indicate that a belief holds in the precondition, or should be added in the postcondition and $-\phi$ to indicate that a belief does not hold (precondition) or should be removed (postcondition). Actions a are modeled by predicates of the form `does(A,Act)` in which `Act` is a term denoting the action and `A` denotes the name of the agent performing it. The dining agents, for example, can perform actions to pick up and put down their (left and right) sticks and eat. The effect rules defining these actions are listed in lines 6-54[2]. An agent can only pickup a stick if the stick is on the table (e.g. lines 7-9 defining the action of picking up a right stick), can only put down a stick when it is holding it (e.g. line 11 defining the action of putting down a right stick) and can eat when it has lifted both

[1] Where `%` is arithmetic modulus.

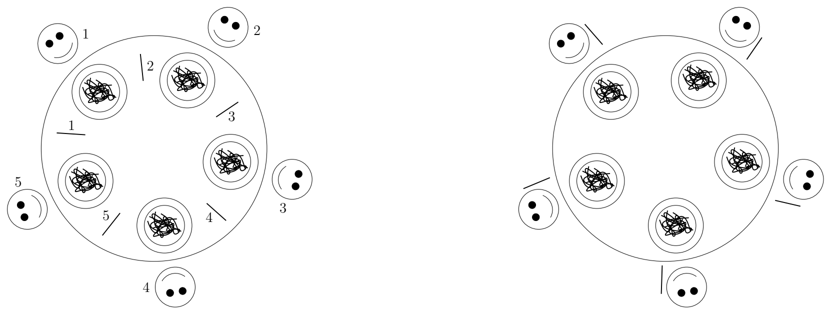
[2] It should be noted that the current ORWELL prototype has limited ability to reason about arithmetic in rule preconditions. Hence the unecessary proliferation of some rules in this example.

Code fragment 2.1 Dining agents implemented in ORWELL.

```
:Brute Facts:
down(1) down(2) down(3) down(4) down(5)
food(1) food(2) food(3) food(4) food(5)
agent(ag1,1) agent(ag2,2) agent(ag3,3) agent(ag4,4) agent(ag5,5)

:Effect Rules:
{agent(A,I), down(I)}
   does(A,pur)
{−down(I), hold(I,r), return(u)}

{agent(A,I), −down(I)} does(A,pur) {return(d)}

{agent(A, I), hold(I,r)} does(A,pdr) {down(I), −hold(I,r)}

{agent(ag1,1), down(2)}
   does(ag1,pul) {−down(2), hold(1,l), return(u)}
{agent(ag1,1), −down(2)} does(ag1,pul) {return(d)}

{agent(ag2,2), down(3)}
   does(ag2,pul) {−down(3), hold(2,l), return(u)}
{agent(ag2,2), −down(3)} does(ag2,pul) {return(d)}

{agent(ag3,3), down(4)}
   does(ag3,pul) {−down(4), hold(3,l), return(u)}
{agent(ag3,3), −down(4)} does(ag3,pul) {return(d)}

{agent(ag4,4), down(5)}
   does(ag4,pul) {−down(5), hold(4,l), return(u)}
{agent(ag4,4), −down(5)} does(ag4,pul) {return(d)}

{agent(ag5,5), down(1)}
   does(ag5,pul) {−down(1), hold(5,l), return(u)}
{agent(ag5,5), −down(1)} does(ag5,pul) {return(d)}

{agent(ag1,1), hold(1,l)}
   does(ag1,pdl) {down(2), −hold(1,l)}

{agent(ag2,2), hold(2,l)}
   does(ag2,pdl) {down(3), −hold(2,l)}

{agent(ag3,3), hold(3,l)}
   does(ag3,pdl) {down(4), −hold(3,l)}

{agent(ag4,4), hold(4,l)}
   does(ag4,pdl) {down(5), −hold(4,l)}

{agent(ag5,5), hold(5,l)}
   does(ag5,pdl) {down(1), −hold(5,l)}

{agent(A,I), hold(I,r), hold(I,l), food(I)}
   does(A,eat)
{−food(I), return(yes)}

{agent(A,I), −food(I)} does(A,eat) {return(no)}
```

Code fragment 2.2 Dining agents implemented in ORWELL(cont.)

```
:CountsAs Rules:
{-hold(1,r), hold(1,l), food(1)} {True} => {viol(1)}
{hold(2,r), -hold(2,l), food(2)} {True} => {viol(2)}
{-hold(3,r), hold(3,l), food(3)} {True} => {viol(3)}
{hold(4,r), -hold(4,l), food(4)} {True} => {viol(4)}
{-hold(5,r), hold(5,l), food(5)} {True} => {viol(5)}
{agent(A,I),-food(I),-hold(I,r),-hold(I,l)} {True} => {reward(I)}

:Sanction Rules:
{viol(A)} => {-food(A), punished(A)}
{reward(A)} => {food(A), rewarded(A)}
```



(a) The initial table arrangement.

(b) A deadlock situation.

Fig. 2. The dining agents problem

sticks and has food on its plate (lines 50-52). Actions might have different effects depending on the particular brute state. To inform agents about the effect of an action we introduce special designated unary facts starting with predicate `return` to pass back information (terms) to the agent performing the action. These facts are not asserted to the brute state. Picking up a stick will thus return `u` (up) in case the stick is successfully lifted (line 9) and `d` (down) otherwise (e.g. line 11). Similarly, the succes of performing an eat action is indicated by returning `yes` (line 52) or `no` (line 54). Note that we assume that agents will only perform the eat action in case they have lifted their stick. Ways for returning information (and handling failure) were not originally described in [9] and are left for future research.

When every agent has decided to eat, holds a left stick and waits for a right stick, we have a deadlock situation (see figure 2b for a graphical representation). One (of many) possible solutions to prevent deadlocks is to implement a protocol in which the odd numbered agents are supposed to pick-up their right stick first and the even numbered agents their left. Because we cannot make any assumptions about the internals of the agents we need to account for the sub-ideal situation in which an agent does not follow the protocol. To motivate the agents to abide by the protocol we implement norms to detect undesirable (violations) and desirable behaviour (code fragment 2.2). The norms in our framework take on the form of elementary counts-as rules relating a set of brute facts with a set of institutional facts (the normative judgment). The rules listed in lines 2, 4 and 6 state that a situation in which an odd numbered agent holds her left stick and not her right while there is food on her plate counts as a violation. Rules listed in lines 3 and 5 implement the symmetric case for even numbered agents. The last rule marks a state in which an agent puts down both sticks when there is no food on her plate as good behaviour. It is important to emphasize that in general hard-wiring the protocol by the action specification (in this case effect rules) such that violations are not possible severely limits the agent's autonomy [2]. It should also be noted that the antecedent of a counts-as rule can also contain institutional facts (in this example these are irrelevant and the institutional precondition is `True`).

Undesirable behaviour is punished and good behaviour is rewarded. This is expressed by the sanction rules (lines 9-11) of code fragment 2.2. Sanction rules are expressed as a kind of inverted counts-as rules relating a set of institutional facts with a set of brute facts to be accommodated in the brute state. Bad behaviour, that is not abiding by the protocol, is thus punished by taking away the food of the agent such that it cannot successfully perform the eat action. Good behaviour, i.e. not unnecesarily keeping hold of sticks, is rewarded with food.

3 The MCAPL Framework for Model Checking Agent Programming Languages

The MCAPL framework is intended to provide a uniform access to model-checking facilities to programs written in a wide range of BDI-style agent programming languages. The framework is outlined in [10] and described in more detail in [3].

Fig. 3 shows an agent executing within the framework. A program, originally programmed in some agent programming language and running within the MCAPL Framework is represented. It uses data structures from the Agent Infrastructure Layer (AIL) to store its internal state comprising, for instance, an agent's belief base and a rule library. It also uses an interpreter for the agent programming language that is built using AIL classes and methods. The interpreter defines the reasoning cycle for the agent programming language which interacts with a model checker, essentially notifying it when a new state is reached that is relevant for verification.

The Agent Infrastructure Layer (AIL) toolkit was introduced as a uniform framework [11] for easing the integration of new languages into the existing execution and verification engine. It provides an effective, high-level, basis for implementing operational semantics [16] for BDI-like programming languages. An operational semantics describes the behavior of a programming language in terms of transitions between program configurations. A configuration describes a state of the program and a transition is a transformation of one configuration γ into another configuration γ', denoted by $\gamma \to \gamma'$. The transitions that can be derived for a programming language are defined by a set of derivation rules of the form $\frac{P}{\gamma \to \gamma'}$ with the intuitive reading that transition $\gamma \to \gamma'$ can be derived when premise P holds. An execution trace in a transition system is then a sequence of configurations that can be generated by applying transition rules to an initial configuration. An execution thus shows a possible behavior of the system at

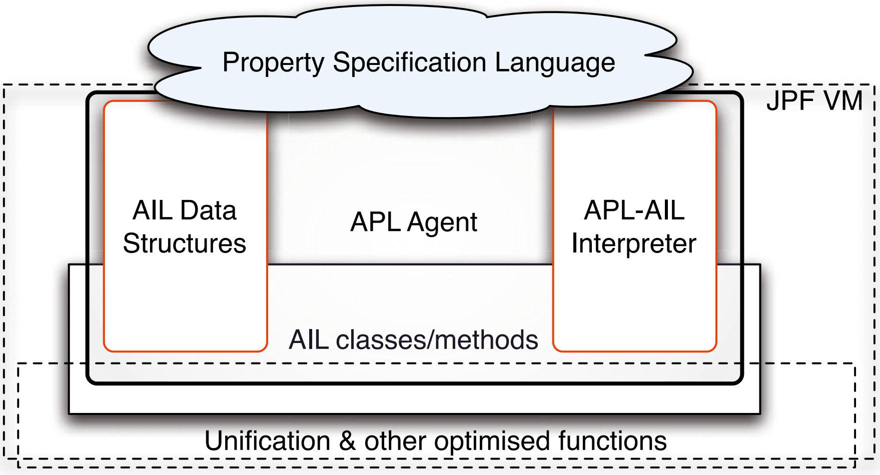


Fig. 3. Outline of Approach

hand. All possible executions for an initial configuration show the complete behavior. The key *operations* of many (BDI-)languages together with a set of standard transition rules form the AIL *toolkit* that can be used by any agent programming language in its own AIL-based interpreter. Of course, it is possible to add custom rules for specific languages.

The agent system runs in the Java Pathfinder (JPF) virtual machine. This is a JAVA virtual machine specially designed to maintain backtrack points and explore, for instance, all possible thread scheduling options (that can affect the result of the verification) [22]. Agent JPF (AJPF) is a customisation of JPF that is optimised for AIL-based interpreters. Common to all language interpreters implemented using the AIL are the AIL-agent data structures for beliefs, intentions, goals, etc., which are accessed by the model checker and on which the modalities of a property specification language are defined. For instance the belief modality of the property specification language is defined in terms of the way logical consequence is implemented within the AIL.

The AIL can be viewed as a platform on which agents programmed in different programming languages co-exist. Together with AJPF this provides uniform model checking techniques for various agent-oriented programming languages and even allows heterogeneous settings [11].

4 Modified Semantics for ORWELL for Implementation in the AIL

In this work we apply the MCAPL framework to the ORWELL language and experiment with the model checking of organisations. Although ORWELL is an organisational language rather than an agent programming language many of its features show a remarkable similarity to concepts that are used in BDI agent programming languages. The brute and insitutional facts, for example, can be viewed as knowledge bases. The belief bases of typical BDI agent languages, which are used to store the beliefs of an agent, are also knowledge bases. Further, the constructs used in modelling effects, counts-as and sanctions are all types of rules that show similarities with planning rules used by agents. This made it relatively straightforward to model ORWELL in the AIL.

The AIL framework assumes that agents in an agent programming language all possess a *reasoning cycle* consisting of several (≥ 1) stages. Each stage describes a coherent activity of an agent, for example, generating plans for achieving goals and acting by executing these plans. Moreover, each stage is a disjunction of transition rules that define how an agent's state may change during the execution of that stage. Only one stage is active at a time and only rules that belong to that stage will be considered. The agent's reasoning cycle defines how the reasoning process moves from one stage to another. The combined rules of the stages of the reasoning cycle define the operational semantics of that language. The construction of an interpreter for a language involves the implementation of these rules (which in some cases might simply make reference to the pre-implemented rules) and a reasoning cycle.

Standard ORWELL [9] does not explicitly consider a reasoning cycle, but what can be considered its reasoning cycle consists of one single transition rule that describes the organisation's response to actions performed by interacting agents. In words, when an action is received, the application of this transition rule;

1. applies one effect rule,
2. then applies all applicable counts-as rules until no more apply and
3. then applies all applicable sanction rules.

The application of this rule thus performs a sequence of modifications to the agent state which the AIL would most naturally present as separate transitions. We needed to reformulate the original rule as a sequence of transition rules in a new form of the operational semantics and include a step in which the organisation perceived the actions taken by the agents interacting with it. Determining all the effects of applying the counts-as rules, for example, was explained in [9] by the definition of a mathematical closure function which was then used in its single transition rule. Although mathematically correct, such a closure function is too abstract to serve as a basis for an actual implementation and needed to be redefined in terms of transition rules for a natural implementation in the AIL.

Figure 4 shows the reworked reasoning cycle for ORWELL. It starts with a perception phase in which agent actions are perceived. Then it moves through two stages which apply an effect rule (B & C), two for applying counts-as rules (D & E) and two for applying sanction rules (F & G). Lastly there is a stage (H) where the results of actions are returned to the agent taking them.

The splitting of the rule phases into two was dictated by the default mechanisms for applying rules[3] in the AIL, in which a set of applicable rules are first generated and then one is chosen and processed. It would have been possible to combine this process

[3] Called plans in the AIL terminology.

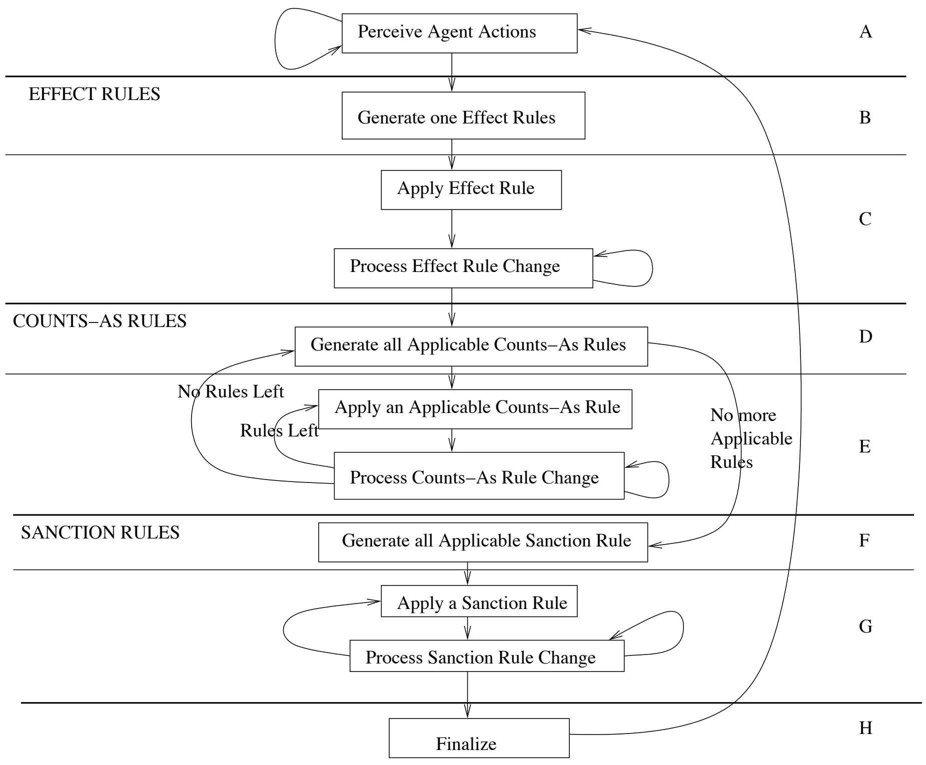

Perceive Agent Actions
A
EFFECT RULES
Generate one Effect Rules
B
Apply Effect Rule
C
Process Effect Rule Change
COUNTS–AS RULES
Generate all Applicable Counts–As Rules
D
No Rules Left
Apply an Applicable Counts–As Rule
Rules Left
No more Applicable Rules
E
Process Counts–As Rule Change
SANCTION RULES
Generate all Applicable Sanction Rule
F
Apply a Sanction Rule
G
Process Sanction Rule Change
H
Finalize

Fig. 4. The ORWELL Reasoning Cycle in the AIL

into one rule, but it was simpler, when implementing this prototype, to leave it in this form, although it complicates the semantics.

Figures 5 to 8 show the operational semantics of ORWELL, reworked for an AIL interpreter and simplified slightly to ignore the effects of unification. The state of an organisation is represented by a large tuple of the form $\langle i, I, BF, IF, ER, CAR, SR, AP, A, RS \rangle$ in which:

- i is the "current intention";
- I is a set of additional "intentions";
- BF is a set of brute facts;
- IF is a set of institutional facts;
- ER is a set of effect rules;
- CAR is a set of counts-as rules;
- SR is a set of sanction rules;
- AP is a set of applicable rules;
- A is a list of actions taken by the agents in the organisation;
- RS is an atomic formula with predicate name `return` for storing the result of the action.

We extend this tuple with one final element to indicate the phase of the reasoning cycle from figure 4 that is currently in focus. This element will always occur as last element of the tuple. In order to improve readability, we show *only* those parts of the agent tuple actually changed or referred to by a transition rule. We use the naming conventions just outlined to indicate which parts of the tuple we refer to, priming the names on the right hand side of the transition where the value has changed. Where there may be confusion we also show their value as an equality – i.e. $i = (a, \epsilon)$ means the current intention is (a, ϵ), if this is changed to $null$ then we will write $i' = null$ on the right hand side of the transition..

The concept of intention is common in many BDI-languages and is used to indicate the *intended means* for achieving a goal or handling an event. Within the AIL, intentions are data structures which associate events with the plans generated to handle that event (including any instantiations of variables appearing in those plans). As plans are executed the intention is modified accordingly so that it only stores that part of the plan yet to be processed. Of course, the concept of intention is not originally used in ORWELL. We slightly abuse this single agent concept to store the instantiated plans associated with any applicable rules. Its exact meaning depends on which type of rule (effect, counts-as or sanction) is considered. When an effect rule is applicable, an intention stores the (unexecuted) postconditions of the rule associated with the action that triggered the rule. When a counts-as or sanction rule is applicable an intention stores its (unexecuted) postconditions together with a record of state that made the rule applicable (essentially the conjunction of its instantiated preconditions). Also the concepts of applicable rules denoting which (effect, counts-as or sanction) rules are applicable (their precondition is satisfied) in a specific situation are AIL specific and are not originally part of ORWELL.

$$\frac{}{\langle i, A = a;A', \mathbf{A}\rangle \rightarrow \langle i' = (a, \epsilon), A', \mathbf{B}\rangle} \qquad (1)$$

Fig. 5. The Operational Semantics for ORWELL as implemented in the AIL (Agent Actions)

Figure 5 shows the semantics for the initial stage. As agents take actions, these are stored in a queue, A, within the organisation for processing[4]. The organisation processes one agent action at a time. The reasoning cycle starts by selecting an action, a, for processing. This is converted into an intention tuple (a, ϵ) where the first part of the tuple stores the action (in this case) which created the intention and the second part of the tuple stores the effects of any rule triggered by the intention, i.e. the brute facts to be asserted and retracted. Initially the effects are indicated by a distinguished symbol ϵ, which indicates that no effects have yet been calculated. We believe that when this rule fires the current intention will be empty (i.e. all its effects will have been processed) but we have not proved this fact.

Figure 6 shows the semantics for processing effect rules. These semantics are very similar to those used for processing counts-as rules and sanction rules and, in many

[4] We use ; to represent list cons.

$$\frac{\{(a, Post) \mid \{Pre\}a\{Post\} \in ER \wedge BF \models Pre\} = \emptyset}{\langle BF, i = (a, \epsilon), AP, \mathbf{B}\rangle \rightarrow \langle BF, i' = \mathbf{null}, AP' = \emptyset, \mathbf{H}\rangle} \tag{2}$$

$$\frac{\{(a, Post) \mid \{Pre\}a\{Post\} \in ER \wedge BF \models Pre\} = AP' \quad AP' \neq \emptyset}{\langle BF, i = (a, \epsilon), AP, \mathbf{B}\rangle \rightarrow \langle BF, i' = (a, \epsilon), AP', \mathbf{C}\rangle} \tag{3}$$

$$\frac{(a, Post) \in AP}{\langle i = (a, \epsilon), AP, \mathbf{C}\rangle \rightarrow \langle i' = (a, Post), AP' = \emptyset, \mathbf{C}\rangle} \tag{4}$$

$$\frac{}{\langle BF, i = (a, +bf;Post), \mathbf{C}\rangle \rightarrow \langle BF' = BF \cup \{bf\}, i' = (a, Post), \mathbf{C}\rangle} \tag{5}$$

$$\frac{}{\langle BF, i = (a, -bf;Post), \mathbf{C}\rangle \rightarrow \langle BF' = BF/\{bf\}, i' = (a, Post), \mathbf{C}\rangle} \tag{6}$$

$$\frac{}{\langle i = (a, []), \mathbf{C}\rangle \rightarrow \langle i' = (a, []), \mathbf{D}\rangle} \tag{7}$$

Fig. 6. The Operational Semantics for ORWELL as implemented in the AIL (Effect Rules)

cases the implementation uses the same code, simply customised to choose from different sets of rules depending upon the stage of the reasoning cycle. Recall that an effect rule is a triple $\{Pre\}\,a\{Post\}$ consisting of a set of preconditions Pre, an action a taken by an agent and a set of postconditions $Post$.

If the action matches the current intention and the preconditions hold , written $BF \models Pre$ (where BF are the brute facts of the organisation), then the effect rule is applicable. Rule 2 pertains to the case in which no effect rule can be applied. This could happen when no precondition is satisfied or if the action is simply undefined. The brute state will remain unchanged, so there is no need for normatively assessing it. Therefore, the organisation cycles on to stage **H** were an empty result will be returned. Applicable effect rules are stored in the set of applicable rules AP (rule 3), of which one applicable rule is chosen (rule 4) and its postconditions are processed (rules 5 and 6). The postconditions consist of a stack of changes to be made to the brute facts, $+bf$ indicates that the fact bf should be added and $-bf$ indicates that a fact should be removed. These are processed by rules 5 and 6 in turn until no more postconditions apply (rule 7). Then it moves on to the next stage (stage **D**) in which the resulting brute state is normatively assessed by the counts-as rules.

Figure 7 shows the semantics for handling counts-as rules. These are similar to the semantics for effect rules except that the closure of all counts-as rules are applied. The set G, is used to track the rules that have been applied. All applicable counts as rules are made into intentions, these are selected one at a time and the rule postconditions are processed. As mentioned before, a counts-as rule may contain institutional facts in its precondition. Thus the application of a counts-as rule might trigger another counts-as rule that was not triggered before. Therefore, when all intentions are processed the stage returns to stage **D**, in order to see if any new counts-as rules have become applicable.

Figure 8 shows the rules governing the application of sanction rules. These are similar to the application of counts-as rules however, since sanction rules consider only

$$\frac{\{(\bigwedge \mathit{Pre}, \mathit{Post}) \mid \{\mathit{Pre}\} \Rightarrow \{\mathit{Post}\} \in \mathit{CAR}/G \wedge \mathit{BF} \cup \mathit{IF} \models \mathit{Pre}\} = \emptyset}{\langle \mathit{BF}, \mathit{IF}, AP, G, \mathbf{D}\rangle \rightarrow \langle \mathit{BF}, \mathit{IF}, AP' = \emptyset, G' = \emptyset, \mathbf{F}\rangle} \tag{8}$$

$$\frac{\{(\bigwedge \mathit{Pre}, \mathit{Post}) \mid \{\mathit{Pre}\} \Rightarrow \{\mathit{Post}\} \in \mathit{CAR}/G \wedge \mathit{BF} \cup \mathit{IF} \models \mathit{Pre}\} = AP' \quad AP' \neq \emptyset}{\langle \mathit{BF}, \mathit{IF}, AP, G, \mathbf{D}\rangle \rightarrow \langle \mathit{BF}, \mathit{IF}, AP', G' = AP' \cup G, \mathbf{E}\rangle} \tag{9}$$

$$\frac{AP \neq \emptyset}{\langle \mathit{org}, I, AP, \mathbf{E}\rangle \rightarrow \langle \mathit{org}, I' = AP \cup I, AP' = \emptyset, \mathbf{E}\rangle} \tag{10}$$

$$\frac{}{\langle \mathit{org}, i = (\bigwedge \mathit{Pre}, []), I = i';I', \mathbf{E}\rangle \rightarrow \langle \mathit{org}, i', I', \mathbf{E}\rangle} \tag{11}$$

$$\frac{}{\langle \mathit{org}, \mathit{IF}, i = (\bigwedge \mathit{Pre}, +\mathit{if};\mathit{Post}), \mathbf{E}\rangle \rightarrow \langle \mathit{org}, \mathit{IF}' = \mathit{IF} \cup \{\mathit{if}\}, i' = (\bigwedge \mathit{Pre}, \mathit{Post}), \mathbf{E}\rangle} \tag{12}$$

$$\frac{}{\langle \mathit{org}, \mathit{IF}, i = (\bigwedge \mathit{Pre}, -\mathit{if};\mathit{Post}), \mathbf{E}\rangle \rightarrow \langle \mathit{org}, \mathit{IF}' = \mathit{IF}/\{\mathit{if}\}, i' = (\bigwedge \mathit{Pre}, \mathit{Post}), \mathbf{E}\rangle} \tag{13}$$

$$\frac{I = \emptyset}{\langle \mathit{org}, i = (\bigwedge \mathit{Pre}, []), I, \mathbf{E}\rangle \rightarrow \langle \mathit{org}, i' = (\bigwedge \mathit{Pre}, []), I, \mathbf{D}\rangle} \tag{14}$$

Fig. 7. The Operational Semantics for ORWELL as implemented in the AIL (Counts-As Rules)

$$\frac{\{(\bigwedge \mathit{Pre}, \mathit{Post}) \mid \{\mathit{Pre}\} \Rightarrow \{\mathit{Post}\} \in \mathit{SR} \wedge \mathit{IF} \models \mathit{Pre}\} = \emptyset}{\langle \mathit{IF}, I, AP, \mathbf{F}\rangle \rightarrow \langle \mathit{IF}, I' = \emptyset, \mathbf{H}\rangle} \tag{15}$$

$$\frac{\{(\bigwedge \mathit{Pre}, \mathit{Post}) \mid \{\mathit{Pre}\} \Rightarrow \{\mathit{Post}\} \in \mathit{SR} \wedge \mathit{IF} \models \mathit{Pre}\} = AP' \quad AP' \neq \emptyset}{\langle \mathit{IF}, AP, \mathbf{F}\rangle \rightarrow \langle \mathit{IF}, AP', \mathbf{G}\rangle} \tag{16}$$

$$\frac{AP \neq \emptyset}{\langle I, AP, \mathbf{G}\rangle \rightarrow \langle I' = AP \cup I, AP' = \emptyset, \mathbf{G}\rangle} \tag{17}$$

$$\frac{}{\langle i = (\bigwedge \mathit{Pre}, []), I = i';I', \mathbf{G}\rangle \rightarrow \langle i', I', \mathbf{G}\rangle} \tag{18}$$

$$\frac{}{\langle \mathit{BF}, i = (\bigwedge \mathit{Pre}, +\mathit{bf};\mathit{Post}), \mathbf{G}\rangle \rightarrow \langle \mathit{BF}' = \mathit{BF} \cup \{\mathit{bf}\}, i' = (\bigwedge \mathit{Pre}, \mathit{Post}), \mathbf{G}\rangle} \tag{19}$$

$$\frac{}{\langle \mathit{BF}, i = (\bigwedge \mathit{Pre}, -\mathit{bf};\mathit{Post}), \mathbf{G}\rangle \rightarrow \langle \mathit{BF}' = \mathit{BF}/\{\mathit{bf}\}, i' = (\bigwedge \mathit{Pre}, \mathit{Post}), \mathbf{G}\rangle} \tag{20}$$

$$\frac{I = \emptyset}{\langle i = (\bigwedge \mathit{Pre}, []), I, \mathbf{G}\rangle \rightarrow \langle i = (\bigwedge \mathit{Pre}, []), I, \mathbf{H}\rangle} \tag{21}$$

Fig. 8. The Operational Semantics for ORWELL as implemented in the AIL (Sanction Rules)

$$\frac{return(X) \in BF \quad RS = []}{\langle org, BF, RS, \mathbf{H}\rangle \rightarrow \langle org, BF' = BF/\{\mathbf{return(X)}\}, RS' = [\mathbf{X}], \mathbf{A}\rangle} \tag{22}$$

$$\frac{return(X) \notin BF \quad RS = []}{\langle org, BF, RS, \mathbf{H}\rangle \rightarrow \langle org, BF, RS' = [\mathbf{none}], \mathbf{A}\rangle} \tag{23}$$

Fig. 9. The Operational Semantics for ORWELL as implemented in the AIL (Finalise)

institutional facts and alter only brute facts there is no need to check for more applicable rules once they have all applied.

Lastly, figure 9 shows the rules of the final stage. The final stage of the semantics returns any results derived from processing the agent action. It does this by looking for a term of the form $return(X)$ in the Brute Facts and placing that result, X, in the result store. The result store is implemented as a blocking queue, so, in this implementation, the rules wait until the store is empty and then place the result in it. When individual agents within the organisation take actions these remove a result from the store, again waiting until a result is available.

Many of these rules are reused versions of customisable rules from the AIL toolkit. For instance the AIL mechanims for selecting applicable "plans" were easily customised to select rules and was used in stages $\mathbf{B}, \mathbf{D}$ and $\mathbf{F}$. Similarly we were able to use AIL rules for adding and removing beliefs from an agent belief base to handle the addition and removal of brute and institutional facts. We modeled ORWELL's fact sets as belief bases and extended the AIL's belief handling methods to deal with the presence of multiple belief bases.

It became clear that the ORWELL stages couldn't be simply presented as a cycle. In some cases we needed to loop back to a previous stage. We ended up introducing rules to control phase changes explicitly (e.g. rule (21)) but these had to be used via an awkward implementational mechanism which involved considering the rule that had last fired. In future we intend to extend the AIL with a generic mechanism for doing this.

It was outside the scope of our exploratory work to verify that the semantics of ORWELL, as implemented in the AIL, conformed to the standard language semantics as presented in [9]. However our aim is to discuss the verification of normative organisational programs and this implementation is sufficient for that, even if it is not an exact implementation of ORWELL.

5 Model Checking Normative Agent Organisations

We implemented the ORWELL Organisation for the dining philosophers system shown in code fragment 2.1 but modified, for time reasons, to consider only three agents rather than five. We integrated this organisation into three multi-agent systems.

The first system (System A) consisted of three agents implemented in the GOAL language. Part of the implementation of one of these agents is shown in code fragment 5.1. This agent has a goal to have eaten (line 4), but initially believes it has not eaten (line 7). It also believes that its left and right stick are both down on the table (also line 7). The agent has capabilities (lines 9-14) to perform all actions provided by the organisation.

Code fragment 5.1 A protocol abiding GOAL agent.

```
:name: ag1                                                    1
                                                              2
:Initial Goals:                                               3
eaten(yes)                                                    4
                                                              5
:Initial Beliefs:                                             6
eaten(no) left(d) right(d)                                    7
                                                              8
:Capabilities:                                                9
pul pul {True} {-left(d), left(R)}                           10
pur pur {True} {-right(d), right(R)}                         11
pdl pdl {True} {-left(u), left(d)}                           12
pdr pdr {True} {-right(u), right(d)}                         13
eat eat {True} {-eaten(no), eaten(R)}                        14
                                                             15
:Conditional Actions:                                        16
G eaten(yes), B left(d), B right(d) |> do(pur)               17
G eaten(yes), B left(d), B right(u) |> do(pul)               18
G eaten(yes), B left(u), B right(u) |> do(eat)               19
B eaten(yes), B left(u) |> do(pdl)                           20
B eaten(yes), B right(u) |> do(pdr)                          21
```

The return value of the organisation is accessed through the special designated variable term R that can be used in the postcondition of the capability specification. The beliefs of the agent will thus be updated with the effect of the action. The conditional actions define what the agent should do in achieving its goals and are the key to a protocol implementation. Whenever the agent has a goal to have eaten and believes it has not to have lifted either stick it will start by picking up its right stick first (line 17). Then it will pick up its left (line 18) and start eating when both are acquired (line 19). Note that if the eat action is successfully performed the agent has accomplished its goal. When the agent believes it has eaten and holds its sticks it will put them down again (lines 20 and 21). Other protocol abiding agents are programmed in a similar fashion provided that ag2 will pick up their left stick first instead of their right. Our expectation was, therefore, that this multi-agent system would never incur any sanctions within the organisation.

System B used a similar set of three GOAL agents, only in this case all three agents were identical (i.e. they would all pick up their right stick first). We anticipated that this group of agents would trigger sanctions.

Lastly, for System C, we implemented three Black Box agents which performed the five possible actions almost at random[5]. The random agents could take no more than five actions in a run of the program, though actions could be repetitions of previous

[5] In order to reduce search we constrained the agents a little internally, so that they could not perform a put down action before a pick up action, and they couldn't eat until after they had performed both pick up actions. The agents had no perceptions of the outside world and so the actions were not necessarily successful.

ones. This system did not conform to the assumption that once an agent has picked up a stick it will not put it down until it has eaten.

We investigated the truth of three properties evaluated on these three multi-agent systems. In what follows $\square$ is the LTL operator, always. Thus $\square\phi$ means that ϕ holds in all states contained in every run of the system. $\diamond$ is the LTL operator, eventually or finally. $\diamond\phi$ means that ϕ holds at some point in every run of a system. The modal operator $\mathcal{B}(ag, \phi)$ stands for "ag believes ϕ" and is used by AJPF to interrogate the knowledge base of an agent. In the case of ORWELL this interrogates the fact bases.

Property 1 states that it is always the case that if the organisation believes (i.e. stores as a brute fact in its knowledge base) all agents are holding their right stick (or all agents are holding their left stick) – i.e., the system is potentially in a deadlock – then at least one agent believes it has eaten (i.e., one agent is about to put down it's stick and deadlock has been avoided).

$$\square((\bigwedge_i \mathcal{B}(org, hold(i,r)) \vee \bigwedge_i \mathcal{B}(org, hold(i,l))) \Rightarrow \bigvee_i \mathcal{B}(ag_i, eaten(yes))) \quad (24)$$

Property 2 states that it is not possible for any agent which has been punished to be given more food.

$$\square \bigwedge_i \neg(\mathcal{B}(org, punished(i)) \wedge \mathcal{B}(org, food(i))) \quad (25)$$

Property 3 states after an agent violates the protocol it either always has no food or it gets rewarded (for putting its sticks down). This property was expected to hold for all systems irrespective of whether the agents wait until they have eaten before putting down their sticks or not.

$$\begin{array}{c}\square \bigwedge_i(\mathcal{B}(org, hold(i,l)) \wedge \neg\mathcal{B}(org, hold(i,r))) \\ \Longrightarrow \\ (\square\neg\mathcal{B}(org, food(i)) \vee \diamond\mathcal{B}(org, rewarded(i)))\end{array} \quad (26)$$

The results of model checking the three properties on the three systems are shown below. We give the result of model checking together with the time taken in hours (h), minutes (m) or seconds (s) as appropriate and the number of states (st) generated by the model checker:

	System A	System B	System C
Property 1	True (40m, 8214 st)	False (2m, 432st)	False (16s, 46st)
Property 2	True (40m, 8214st)	True (30m, 5622st)	False (11s, 57st)
Property 3	True (1h 7m , 9878st)	True (1h 2m, 10352st)	True (15h, 256049 st)

It should be noted that transitions between states within AJPF generally involve the execution of a considerable amount of JAVA code in the JPF virtual machine since the system only branches the search space when absolutely necessary. There is scope, within the MCAPL framework for controlling how often properties are checked. In our case we had the properties checked after each full execution of the ORWELL reasoning cycle. This was a decision made in an attempt to reduce the search space further. So in some cases above a transition between two states represents the execution of all the rules from stages **A** to **H** of the ORWELL reasoning cycle. Furthermore the JPF virtual machine is slow, compared to standard JAVA virtual machines, partly because of the extra burden it incurs maintaining the information needed for model checking. This accounts for the comparatively small number of states examined for the time taken when these results are compared with those of other model checking systems. Even though we excluded as much as possible of the internal state of our random agents there was clearly a much larger search space associated with them. We attribute this to the much higher number of "illogical" states that occur - (when an agent tries to perform an impossible action). We believe it likely that verifying an organisation containing agents with known internal states will prove considerably more computationally tractable than verifying organisations that contain entirely random agents.

In the process of conducting this experiment we discovered errors, even in the small program we had implemented. For instance we did not, initially, return a result when an agent attempted to pick up a stick which was held by another agent. This resulted in a failure of the agents to instantiate the result variable and, in some possible runs, to therefore assume that they had the stick and to attempt to pick up their other stick despite that being a protocol violation. This showed the benefit of model checking an organisation with reference to agents that are assumed to obey its norms.

The experiments also show the benefits of allowing access to an agent's state when verifying an organisation in order to, for instance, check that properties hold under assumptions such as that agents do not put down sticks until after they have eaten. The more that can be assumed about the agents within an organisation the more that can be proved and so the behaviour of the organisation with respect to different kinds of agent can be determined.

6 Conclusions

In this paper we have explored the verification of multi-agent systems running within a normative organisation. We have implemented a normative organisational language, ORWELL, within the MCAPL framework for model checking multi-agent systems in a fashion that allows us to model check properties of organisations.

We have investigated a simple example of an organisational multi-agent system based on the dining philosophers problem and examined its behaviour in settings where we make very few assumptions about the behaviour of the agents within the system and in settings where the agents within the system are white box (i.e., the model checker has full access to their internal state). We have been able to use these systems to verify properties of the organisation, in particular properties about the way in which the organisation handles norms and sanctions.

An interesting result of these experiments has been showing that the use of white box agents allows us to prove a wider range of properties about the way in which the organisation behaves with respect to agents that obey its norms, or agents that, even if they do not obey its norms, respect certain assumptions the organisation embodies about their operation. In particular the white box system enabled us to detect a bug in the organisational code which revealed that the organisation did not provide agents which did obey its norms with sufficient information to do so. This bug would have been difficult to detect in a system where there was no information about the internal state of the constituent agents, since the property that revealed it did not hold in general.

In more general terms the verification of organisations containing white box agents enables the verification that a given multi-agent system respects the norms of an organisation.

References

1. Aştefănoaei, L., Dastani, M., Meyer, J.-J., Boer, F.S.: A verification framework for normative multi-agent systems. In: Bui, T.D., Ho, T.V., Ha, Q.T. (eds.) PRIMA 2008. LNCS (LNAI), vol. 5357, pp. 54–65. Springer, Heidelberg (2008)
2. Aldewereld, H.: Autonomy versus Conformity an Institutional Perspective on Norms and Protocols. PhD thesis, Utrecht University, SIKS (2007)
3. Bordini, R.H., Dennis, L.A., Farwer, B., Fisher, M.: Automated Verification of Multi-Agent Programs. In: Proc. 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 69–78 (2008)
4. Bordini, R.H., Fisher, M., Visser, W., Wooldridge, M.: Model Checking Rational Agents. IEEE Intelligent Systems 19(5), 46–52 (2004)
5. Bordini, R.H., Fisher, M., Visser, W., Wooldridge, M.: Verifying Multi-Agent Programs by Model Checking. Journal of Autonomous Agents and Multi-Agent Systems 12(2), 239–256 (2006)
6. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak Using Jason. Wiley Series in Agent Technology. John Wiley & Sons, Chichester (2007)
7. Cliffe, O., Vos, M.D., Padget, J.A.: Answer set programming for representing and reasoning about virtual institutions. In: Inoue, K., Satoh, K., Toni, F. (eds.) CLIMA 2006. LNCS (LNAI), vol. 4371, pp. 60–79. Springer, Heidelberg (2007)
8. Dastani, M.: 2APL: a practical agent programming language. Autonomous Agents and Multi-Agent Systems 16(3), 214–248 (2008)
9. Dastani, M., Tinnemeier, N.A.M., Meyer, J.-J.C.: A programming language for normative multi-agent systems. In: Dignum, V. (ed.) Multi-Agent Systems: Semantics and Dynamics of Organizational Models, ch. 16. IGI Global (2008)
10. Dennis, L.A., Farwer, B., Bordini, R.H., Fisher, M.: A Flexible Framework for Verifying Agent Programs. In: Proc. 7th International Conference on Autonomous Agents and Multiagent Systems (AAMAS). ACM Press, New York (2008) (Short paper)
11. Dennis, L.A., Fisher, M.: Programming verifiable heterogeneous agent systems. In: Hindriks, K.V., Pokahr, A., Sardina, S. (eds.) ProMAS 2008. LNCS, vol. 5442, pp. 27–42. Springer, Heidelberg (2009)
12. Grossi, D.: Designing Invisible Handcuffs. Formal Investigations in Institutions and Organizations for Multi-agent Systems. PhD thesis, Utrecht University, SIKS (2007)
13. Hindriks, K.V., de Boer, F.S., van der Hoek, W., Meyer, J.-J.C.: Agent programming with declarative goals. In: Castelfranchi, C., Lespérance, Y. (eds.) ATAL 2000. LNCS (LNAI), vol. 1986, pp. 228–243. Springer, Heidelberg (2001)
14. Huguet, M.-P., Esteva, M., Phelps, S., Sierra, C., Wooldridge, M.: Model checking electronic institutions. In: MoChArt 2002, pp. 51–58 (2002)
15. Kacprzak, M., Lomuscio, A., Penczek, W.: Verification of Multiagent Systems via Unbounded Model Checking. In: Proc. 3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 638–645. IEEE Computer Society, Los Alamitos (2004)

16. Plotkin, G.D.: A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus (1981)
17. Ricci, A., Viroli, M., Omicini, A.: Give agents their artifacts: the A&A approach for engineering working environments in MAS. In: AAMAS (2007)
18. Searle, J.R.: The Construction of Social Reality. Free Press, New York (1995)
19. Shoham, Y.: Agent-oriented programming. AI 60(1), 51–92 (1993)
20. Vázquez-Salceda, J., Aldewereld, H., Grossi, D., Dignum, F.: From human regulations to regulated software agents' behavior. AI & Law 16(1), 73–87 (2008)
21. Viganò, F.: A framework for model checking institutions. In: Edelkamp, S., Lomuscio, A. (eds.) MoChArt IV 2006. LNCS (LNAI), vol. 4428, pp. 129–145. Springer, Heidelberg (2007)
22. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model Checking Programs. Automated Software Engineering 10(2), 203–232 (2003)