

RUN-TIME MODEL CHECKING OF INTERACTION AND DEONTIC MODELS FOR MULTI-AGENT SYSTEMS

Nardine Osman David Robertson Christopher Walton

School of Informatics, The University of Edinburgh, Edinburgh EH8 9LE, UK

Abstract

This paper is concerned with the problem of obtaining predictable interactions between groups of agents in open environments when individual agents do not expose their BDI logic. The most popular approaches to this in practise have been to model interaction protocols and to model the deontic constraints imposed by individual agents. Both of these approaches are appropriate and necessary but their combination creates the practical problem of ensuring that interaction protocols are meshed with agents that possess compatible deontic constraints. This is essentially an issue of property checking dynamically at run-time. We show how model checking can be applied to this problem.

1 Introduction

The key to the development of open multi-agent systems (MAS) is the ability to provide reliable, predictable interaction. However, the nondeterministic nature of these systems results in difficulties in the verification process. MAS are open systems in which different agents might join/leave a system at any time. Furthermore, a key concept of MAS is the autonomy of its agents. Agents' actions are based on their own beliefs and goals. The lack of an explicitly predefined set of agents makes it difficult to verify such systems as opposed to verifying traditional hardware or software systems.

Interactions — the backbone that holds a multi-agent system together — may be affected by at least the following: (a) interaction rules for coordinating messages between agents, (b) deontic rules for specifying obligations, permissions, and prohibitions, and (c) rules on the agent level for dealing with the agents' knowledge, belief system, desires, goals, intentions, etc. To reflect these distinctions, we employ a three-layered system architecture (Figure 1).

Although an interaction protocol for a given scenario is global, different agents have different deontic rules that affect the collaboration of agents and services in the scenario. An agent might have restrictions on which agents to deal with, what service it can/cannot access, or what actions it can/cannot perform. As a result, an interaction protocol which works flawlessly with one group of agents might break with another. Unfortunately, interaction and deontic models are complex, and behavioural problems need to be checked automatically.

We therefore propose the use of a dynamic model checker for verifying an instance of the interaction protocol: an interaction protocol working with a selected group of agents, or deontic rules. Furthermore, since deontic rules are dynamic — while the interaction protocol may be fixed, deontic rules change with the agents engaged in the interaction — the verification process needs to be done automatically by the agents during interaction time. The model checker is then invoked by agents at run-time when deontic rules are made available.

Our verification technique deliberately neglects the agents' BDI model. This is because we do not believe that agents are likely to make this information, as goals and intentions, public. More importantly, we do not believe that it would be easy for agents to extract such information from

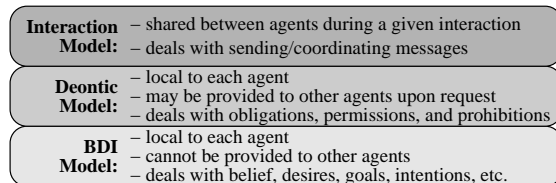


Figure 1: A 3-layered MAS architecture approach

their specification. Usually, this information is not specified declaratively but rather it is implicit in the procedural implementation of the agent. Finally, we are not interested in predetermining how agents will act in response to each action. Instead, we simply try to verify that for a given interaction scenario, given agents are capable of collaborating with no conflicts arising from their combined sets of deontic constraints.

In Section 2, we explain in detail the distinction we make between interaction and deontic definitions, relating those in Section 3 to model checking. Interaction models are shared between agents and are therefore portable. In Section 4, we explain how this is achieved in the LCC language and demonstrate how this links cleanly to the types of process calculi used in model checking. Section 5 then defines the language used to describe properties checked by our system while the model checking algorithm (which is surprisingly compact) is summarised in Section 6.

2 Motivation and Design Goals

Let us consider the following example. For finding and reserving a suitable vacation package consisting of booking a flight ticket and a hotel reservation, a customer agent contacts a broker to find a suitable travel agent. The broker searches for appropriate agents for the given scenario. The scenario is similar to the travel agent use case of [7]. Figure 2 presents an overview of the interaction while Figure 3 defines the section of the interaction while Figure 3 defines the section of the interaction while Figure 3 defines the section of the interaction between the customer and travel agents (I_{CT} of Figure 2).

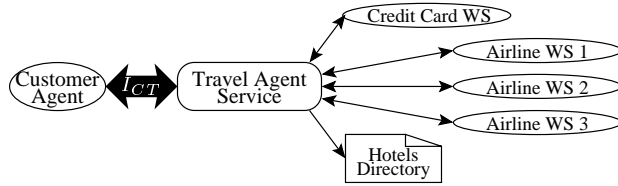


Figure 2: Overview of the travel agent scenario [7]

The interaction starts when a customer agent (C) provides the travel agent (T) with its vacation's start date, end date, and destination (SD, ED, D). The travel agent forwards this information to the airline web services (As) for retrieving quotations (FL) which are forwarded to the customer agent. After the customer selects a flight (Fx), the travel agent searches the hotel directory (HD) and sends a detailed list of hotel options (HL) back to the customer. The customer selects a hotel (Hx), the travel agent computes the total amount (TA) to be paid, and the customer sends its payment details (PD). The travel agent verifies the payment details with the credit card web service (CD), which either provides a signed payment authorisation ($PId, Sign$) or the reason (R)

The interaction starts when a customer agent (C) provides the travel agent (T) with its vacation's start date, end date, and destination (SD, ED, D). The travel agent forwards this information to the airline web services (As) for retrieving quotations (FL) which are forwarded to the customer agent. After the customer selects a flight (Fx), the travel agent searches the hotel directory (HD) and sends a detailed list of hotel options (HL) back to the customer. The customer selects a hotel (Hx), the travel agent computes the total amount (TA) to be paid, and the customer sends its payment details (PD). The travel agent verifies the payment details with the credit card web service (CD), which either provides a signed payment authorisation ($PId, Sign$) or the reason (R)

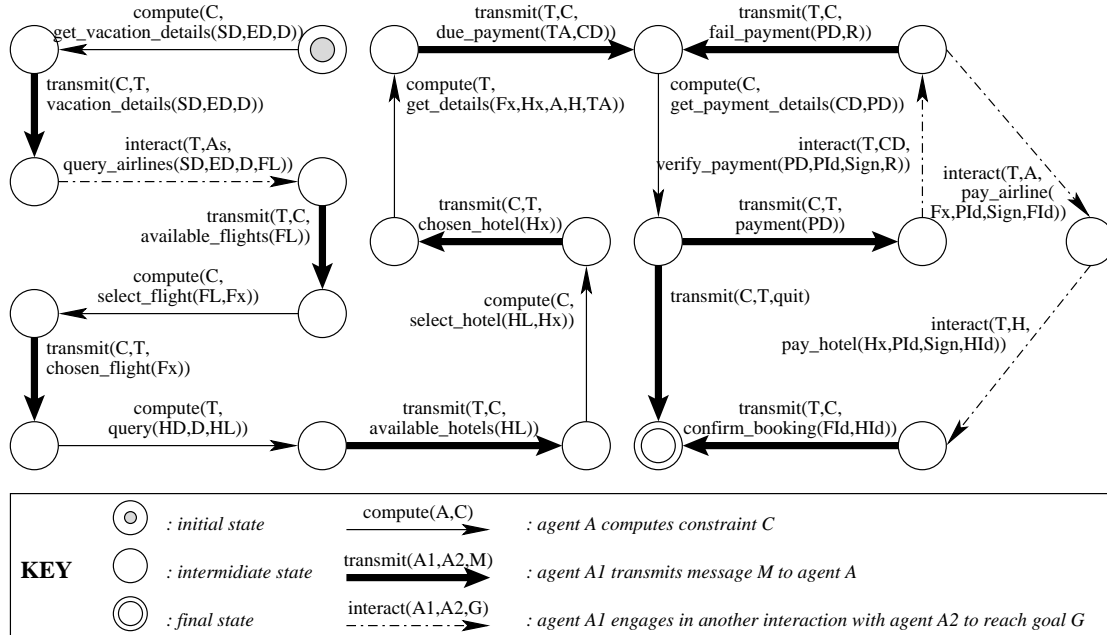


Figure 3: The rules of interaction between the customer and travel agent (I_{CT} of Figure 2)

#	Rule	Enforced by	Description
1-	$(A, D, +query(D, _, _)) \leftarrow (A, D, +access).$	Broker	Broker verifies that an agent is capable of performing a query on a directory by verifying that it has access to the directory
2-	$(A, self, +access) \leftarrow member(A, syta).$	Hotel directory	Hotel directory allows A to access it only if it was a member of $syta$
3-	$(A, _, +get_payment_details(_, CD)) \leftarrow customer(A, CD).$	Broker	Broker verifies that customer agent A is capable of paying its bills by verifying that it is a customer of the selected credit card web service CD
4-	$(CD, _, +authenticate(X.509)).$	Travel agent	Travel agent needs to ensure that the credit card web service CD is capable of authenticating itself with $X.509$ certificate
5-	$(self, _, -encrypt(X)) \leftarrow \neg(X = OpenPGP).$	Customer agent	Customer agent is prohibited to use any encryption other than $OpenPGP$

Table 1: A sample of deontic rules

for payment failure. If the payment is authorised, a copy is sent to the airline and hotel agents (A and H) for confirming the booking. Otherwise, the customer is informed of the failure and it might either choose to retry sending its payment details or quit the interaction.

In addition to the interaction rules (Figure 3), the objects involved may also lay down their own set of restrictions: their deontic rules. Table 1 provides a sample of such rules. For example, the customer agent’s request for booking both a plane ticket and a hotel requires the travel agent to be capable of querying the hotel directory. The broker then needs to verify Rule 1 which states that an agent A may query a directory D only if it has access to it (‘+’ implies an action is permitted while a ‘-’ implies it is prohibited). The success of this rule, however, is dependent on other agents’ deontic rules. For example, the hotel directory may enforce its own rule (Rule 2) which states that an agent A may access it only if it is a member of the Student and Youth Travel Association ($syta$). Deontic rules may be used to address issues such as access control (Rule 1), authorisation (Rule 2), authentication and trust (Rule 4), security (Rule 5), and others (Rule 3). While some of these rules (e.g. Rules 1 and 3) are a requirement for the broker to fulfil, others (e.g. Rules 2, 4 and 5) are a requirement for other agents and services engaged in this scenario (see ‘Enforced by’ column of Table 1). However, it is the broker’s responsibility to make sure that no conflicts arise from the agents’ requirements and constraints, and that the deontic rules of all agents engaged in a given scenario are consistent.

With the broker being responsible for finding suitable agents for a given interaction protocol, it should also be capable of verifying, at interaction time, that the protocol it has instantiated with agents is likely to work. This, however, relies on the correctness of the interaction protocol as well as the compatibility of the chosen agents. For example, trying to ally the customer agent with a travel agent that does not have access to a hotel directory will result in a scenario failure, regardless of whether the interaction protocol itself is error free or not. This requires a verifier that can handle both interaction and deontic constraints, and is capable of operating automatically at run-time. The broker could then use such a verifier to verify an instance of the interaction protocol — the interaction protocol for a given set of agents.

3 Implementation Plan

Our goal is to achieve a verifier which could be used by agents at interaction time for verifying MAS through the verification of the interaction and deontic rules. Figure 4 illustrates the move from the design to the implementation plan.

As illustrated by the travel agency example, the broker agent will need to verify the interaction protocol for various deontic rules until a team of collaborating agents is reached. In open systems consisting of autonomous agents, it is necessary for agents to be ca-

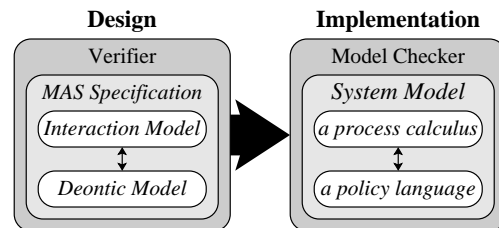


Figure 4: Design and implementation plan

pable of automatically verifying, at run-time, dynamic protocols affected by dynamic deontic rules. For this reason, we choose model checking from amongst other verification techniques because it provides a fully automatic verification process.

For specifying interaction protocols, which deal with coordinating messages between agents, we choose process calculus. Process calculus is a calculus for representing concurrent and distributed processes, and accounts for the non-deterministic and non-terminating nature of these processes. Its success in efficiently describing the rules for coordinating messages makes them especially appealing for specifying interaction protocols of MAS.

Policy languages, on the other hand, have been widely used in hardware systems and networks for expressing deontic rules — the rules of obligations, permissions and prohibitions. Policy languages address issues of security, trust negotiation, access control mechanism, authorisations, etc. This makes them good candidates for specifying agents’ deontic rules.

Implementing the Model Checker The model checking problem can be defined as follows: Given a finite transition system S and a temporal formula ϕ , does S satisfy ϕ ? The model checking process is divided into three stages: modelling, specification, and verification. The system to be verified must first be modelled in the language of the model checker S . The properties to which the system model is verified upon should be specified using the model checker’s temporal logic ϕ . Both the system model and the properties specification are fed to the model checker for the verification stage. The model checker is, essentially, an algorithm that decides whether a model S satisfies a formula ϕ . Some model checkers may also provide a counter example when the property is not satisfied. This is traditionally used to aid the human debugging process. Since humans are not involved in our automatic checking, we do not need our model checker to generate counter examples. Figure 5 provides a representation of the model checking process.

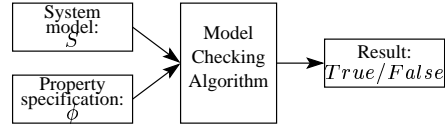


Figure 5: The model checking process

Our system model S is a bundle of interaction and deontic rules (Figure 4). For specifying the interaction rules of the system model, we choose the Lightweight Coordination Calculus (LCC) (Section 4) mainly for two reasons (refer to Section 8 for details): (1) it supports the attachment of a deontic layer to the interaction layer, and (2) it supports the use of a dynamic local model checker. For defining the property specifications ϕ , we choose a modified version of the modal μ -calculus (Section 5) basically for its contribution to the compact size of our dynamic model checker. Finally, a logic-based local model checking algorithm (Section 6) is chosen for implementing our verifier.

The result is a significantly small sized model checker (based solely on the rules of Figures 9 and 12) implemented in tabled Prolog. The use of computational logic provides us with efficiency in computing constraints necessary for verifying deontic constraints. Furthermore, the local model checking technique implies that the state-space is not constructed beforehand, but generated and traversed one step at a time until a solution is reached. The use of tabled Prolog for performing local model checking throws the burden of searching the state-space on the tabled Prolog system keeping the model checker small and simple. This relatively efficient and extremely compact model checker can be used automatically by agents at run-time.

4 Lightweight Coordination Calculus (LCC)

LCC is the calculus used for specifying interaction protocols of MAS. It is based on the concept of agents playing roles and sharing a dialogue framework for achieving distributed coordination. An interesting and important aspect of LCC is its capability to define the interaction protocol without having to specify details of agents involved in this interaction. This supports our requirement for separating the interaction layer from the agent layer. Furthermore, the constraints in the LCC language provide the link needed for connecting the deontic layer to the interaction layer (Section 4.1). The use of LCC, however, has several other advantages. Its syntax presents it as a process calculus which makes it suitable to be used as our model checker’s description language (Section 4.3). The syntax also presents it as a logic programming language [13] which affects the efficiency of our constraints’ computation (Section 8). Finally, LCC is a lightweight calculus whose

only requirement on agents that want to engage in an interaction is to be able to apply the transition rules of Figure 9 (the clause expansion mechanism). This lightweight nature along with its clause expansion mechanism provide the support needed for a dynamic local model checker (Section 6.1).

4.1 LCC Syntax

The LCC interaction framework is the set of clauses specifying the expected message passing behaviour. Its syntax is given in Figure 6.

Agents, in LCC, are defined by their roles and identifiers. A framework is composed of a set of clauses. A clause gives each agent role a definition that specifies its acceptable behaviour. An agent can either do nothing (usually used for internal computations), take a different role, or send/receive messages ($M \Rightarrow A$, $M \Leftarrow A$). Agent definitions can get more complex by using the sequential (*then*), choice (*or*), parallel composition (*par*), and conditional (\leftarrow) operators. The conditional operator is used for linking constraints to message passing actions. These constraints can be used to link the interaction model to the deontic model.

Example: The Travel Agency Scenario To illustrate the specification of systems with LCC, let us consider a section of the travel agency scenario. Figure 7 models the interaction between the customer and the travel agent described earlier in Figure 3. The first two clauses specify the interaction rules of the two roles played by the customer agent. The interaction starts when the agent retrieves its vacation details: start date SD , end date ED , and destination D . It then sends these details to the travel agent, receives a list of available flights FL , selects an appropriate flight Fx , sends its choice to the travel agent, receives a list of available hotel options HL , selects a hotel Hx , sends its choice to the travel agent, receives the bill of amount TA to be paid via credit card CD , and finally takes a different role *paying_customer* for paying its bill. The role *paying_customer* is responsible for retrieving the payment details (e.g. credit card number, expiry date, etc.). Then it either receives a message confirming its

```

Framework := {Clause, ...}
Clause    := Agent :: ADef
Agent     := a(Role, Id)
ADef     := null [ $\leftarrow$  C] | Agent [ $\leftarrow$  C] | Message [ $\leftarrow$  C] |
           ADef then ADef | ADef or ADef |
           ADef par ADef
Message  := M  $\Rightarrow$  Agent | M  $\Leftarrow$  Agent
C        := Term | C  $\wedge$  C | C  $\vee$  C
Role     := Term
M        := Term

```

where, $[X]$ denotes zero or one occurrence of X , $null$ is an event which does not involve message passing, $Term$ is a structured term in Prolog syntax, and Id is either a variable or a unique agent identifier.

Figure 6: Syntax of the LCC dialogue framework

```

a(customer(T), C) ::
  vacation_details(SD, ED, D)  $\Rightarrow$  a(travel_agent(., ., .), T)
   $\leftarrow$  get_vacation_details(SD, ED, D) then
  available_flights(FL)  $\Leftarrow$  a(travel_agent(., ., .), T) then
  chosen_flight(Fx)  $\Rightarrow$  a(travel_agent(., ., .), T)
   $\leftarrow$  select_flight(FL, Fx) then
  available_hotels(HL)  $\Leftarrow$  a(travel_agent(., ., .), T) then
  chosen_hotel(Hx)  $\Rightarrow$  a(travel_agent(., ., .), T)
   $\leftarrow$  select_hotel(HL, Hx) then
  due_payment(TA, CD)  $\Leftarrow$  a(travel_agent(., ., .), T) then
  a(paying_customer(T, CD), C).

a(paying_customer(T, CD), C) ::
  payment(PD)  $\Rightarrow$  a(verify_payment(., ., .), T)
   $\leftarrow$  get_payment_details(CD, PD) then
  (confirm_booking(FId, HId)  $\Leftarrow$  a(verify_payment(., ., .), T)
  or
  (fail_payment(PD, R)  $\Leftarrow$  a(verify_payment(., ., .), T) then
  (a(paying_customer(T, CD), C)
   $\leftarrow$  retry_payment(CD)
  or
  quit  $\Rightarrow$  a(verify_payment(., ., .), T)
   $\leftarrow$   $\neg$ retry_payment(CD) ) ) ).

a(travel_agent([As], HD, CD), T) ::
  vacation_details(SD, ED, D)  $\Leftarrow$  a(customer(., .), C) then
  a(query_airlines([As], SD, ED, D), T) then
  a(get_airline_replies([As], [], FL), T) then
  available_flights(FL)  $\Rightarrow$  a(customer(., .), C) then
  chosen_flight(Fx)  $\Leftarrow$  a(customer(., .), C) then
  null  $\leftarrow$  query(HD, D, HL) then
  available_hotels(HL)  $\Rightarrow$  a(customer(., .), C) then
  chosen_hotel(Hx)  $\Leftarrow$  a(customer(., .), C) then
  due_payment(TA, CD)  $\Rightarrow$  a(customer(., .), C) then
  a(verify_payment(CD, H, A), T)
   $\leftarrow$  get_airline_agent(Fx, A)  $\wedge$  get_hotel_agent(Hx, H)

a(verify_payment(CD, H, A), T) ::
  payment(PD)  $\Leftarrow$  a(paying_customer(., .), C) then
  a(verify_payment(PD, CD, PId, Sign, R), T) then
  ( a(pay_services(H, A, PId, Sign, FId, HId), T)
   $\leftarrow$  R = null then
  confirm_booking(FId, HId)  $\Rightarrow$  a(paying_customer(., .), C) )
  or
  ( fail_payment(PD, R)  $\Rightarrow$  a(paying_customer(., .), C)
   $\leftarrow$   $\neg$ R = null then
  (a(verify_payment(CD, H, A), T)
  or quit  $\Leftarrow$  a(paying_customer(., .), C) ) ).

```

Figure 7: LCC interaction model of Figure 3

bookings (*confirm_booking*), or a message informing it of the reason R for the payment's failure. In the latter case, the agent might either decide to retry its payment ($a(\text{paying_customer}(T, CD), C)$) or send a *quit* message to the travel agent to conclude the interaction.

Similarly, the last two clauses specify the travel agent's rules governing its interaction with the customer. To keep the example simple and short, Figure 7 omits role definitions dealing with the travel agent's interaction with other agents: *query_airlines*, *get_airline_replies*, *pay_services*, etc.

4.2 LCC Clause Expansion

This section explains the clause expansion mechanism of LCC which supports decentralised coordination. The mechanism also directly affects our choice of model checking technique: local model checking (see Section 6.1).

Decentralised coordination is achieved by sending the protocol along with the messages. When an agent needs to send a message to another agent, the tuple $(I, M, A, R, \mathcal{P})$ is transmitted, where I identifies the interaction, M the message, A the receiving agent, R the receiving agent's role in the interaction, and \mathcal{P} the protocol. The protocol itself consists of three elements: a set of LCC clauses P_F that defines the protocol framework, a set of clauses P_S that defines the current protocol state, and a set of clauses K defining the common knowledge. The protocol framework is the original protocol which remains unchanged throughout an interaction. The protocol state consists of those clauses which are constantly modified to keep track of the current protocol state. Common knowledge in LCC is the knowledge needed to carry out a given interaction protocol. It is specific to the given interaction. Please refer to [12] for further details.

Figure 8 describes the algorithm of LCC's coordination mechanism. The algorithm is triggered when an agent receives a tuple of the form $(I, M, A, R, \mathcal{P})$.

Upon receiving a tuple $(I, M, A, R, (P_F, P_S, K))$, the agent checks whether a copy of its own protocol state exists in P_S by checking for a clause matching its role R and Id A . If such a clause does exist, then it is retrieved. Otherwise, the agent's original clause is retrieved from P_F . The incoming message M is added to the list of incoming messages M_i and the transition rules of Figure 9 are applied. The agent's new protocol state replaces the old one (if it existed) in P_S

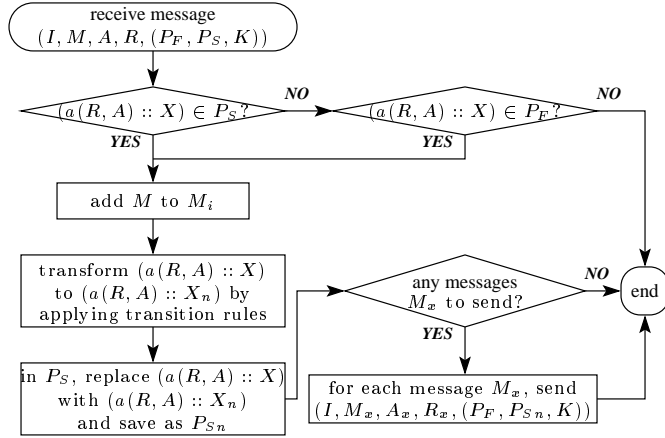


Figure 8: LCC's coordination mechanism

$$\begin{array}{c}
\frac{}{M \leftarrow A \xrightarrow{in(M)} nil} \quad \frac{}{(A \leftarrow C) \xrightarrow{\#(X)} A} \quad \frac{}{sat(C) \wedge X \text{ in } C} \\
\frac{}{M \Rightarrow A \xrightarrow{out(M)} nil} \quad \frac{A \xrightarrow{a} E}{(A \leftarrow C) \xrightarrow{a} E} \quad \frac{}{sat(C) \wedge (a \neq \#/-)} \\
\frac{}{null \xrightarrow{\#} nil} \quad \frac{A \xrightarrow{a} E}{A \text{ par } B \xrightarrow{a} E \text{ par } B} \\
\frac{B \xrightarrow{a} E}{A \xrightarrow{a} E} \quad A :: B \quad \frac{B \xrightarrow{a} E}{A \text{ par } B \xrightarrow{a} A \text{ par } E} \\
\frac{A \xrightarrow{a} E}{A \text{ or } B \xrightarrow{a} E} \quad \frac{A \xrightarrow{a} E \quad B \xrightarrow{\bar{a}} F}{A \text{ par } B \xrightarrow{\tau} E \text{ par } F} \\
\frac{B \xrightarrow{a} E}{A \text{ or } B \xrightarrow{a} E} \quad \frac{A \xrightarrow{a} nil}{A \text{ then } B \xrightarrow{a} B} \\
\frac{A \xrightarrow{a} E}{A \text{ then } B \xrightarrow{a} E \text{ then } B} \quad E \neq nil
\end{array}$$

$sat(C)$ is true if the constraint C can be satisfied.
 nil is the empty process which can not perform any action.
 $in(M)$ is the action of receiving a message M .
 $out(M)$ is the action of sending a message M .
 $\#/-$ is the action of internal computations.
 a is any action (message passing or internal computations).
 \bar{a} is the co-action of a . A message input action is the co-action of its output action, and vice versa. Note that internal computational actions do not have co-actions.
 τ is a complete internal action. It is the result of carrying out an action and its co-action in parallel.
 $X \text{ in } C$ implies that X is a term in the conjunction of terms C .

Figure 9: LCC's transition rules

resulting in a new protocol state P_{Sn} . Finally, messages that need to be sent to other agents will be transmitted via the tuple $(I, M_x, A_x, R_x, (P_F, P_{Sn}, K))$.

For the agent to perform a transition step, the transition rules of Figure 9 are applied exhaustively. The rules state that $M \leftarrow A$ can perform a transition $in(M)$ to the empty process nil by retrieving the incoming message M . $M \Rightarrow A$ can perform a transition $\overline{out}(M)$ to nil by sending the message M . $null$ can perform the transition $\#$ to nil by performing the appropriate internal computations. $A \leftarrow C$ can perform a transition to E if C is satisfied and A can perform a transition to E . A , with definition $A::B$, can perform a transition to E if B can perform a transition to E . A or B can perform a transition to E if either A or B can perform a transition to E . A par B can perform a transition either to E par B if A can perform a transition to E , or to A par E if B can perform a transition to E . A par B can also perform the transition τ to E par F if both A and B can perform transitions to E and F , respectively. Finally, A then B can perform a transition to B if A can perform a transition to the empty process nil ; otherwise, it can perform a transition to E then B if A can perform a transition to E .

4.3 LCC: a Process Calculus for Modelling MAS

LCC — apart from being the calculus used for both specifying the interaction model and building the executable model — is a process calculus. We propose a model checker that accepts LCC as its description language. The LCC protocol, capturing the actual system to be checked, is directly fed to the model checker. This avoids the complexity of modelling the system in another language, and the possibility of introducing errors in doing so. Eliminating this step contributes to the remarkably small size of the model checker.

Comparing LCC to traditional process calculi: an agent in the LCC language is equivalent to a process. The syntax of an LCC process, as defined earlier, is:

$$ADef ::= null[\leftarrow C] \mid Agent[\leftarrow C] \mid Message[\leftarrow C] \mid ADef \text{ then } ADef \mid ADef \text{ or } ADef \mid ADef \text{ par } ADef$$

Similar to traditional process calculi, an agent can be defined in terms of other agents, i.e. an agent can take a different role ($Agent = a(Role, Id)$). The actions an agent can take are restricted to message passing actions. $M \Rightarrow A$ and $M \leftarrow A$ are used for sending and receiving messages, respectively. This is similar to Milner's CCS [10] value passing actions $a(x)$ and $\bar{a}(x)$ where x represents the message sent and a represents the channel between the two communicating agents. However, LCC names the processes involved rather than the channels. The sequential operator *then* is similar to Hoare's CSP's [8] sequential operator $;$ rather than CCS's prefix operator $[\cdot]$. The choice operator *or* is equivalent to $+$ in CCS. Similarly, the parallel composition operator *par* (in the current version of LCC) is equivalent to $||$ in CCS. LCC also defines a conditional operator \leftarrow equivalent to *if C then E* in CCS. LCC's empty process *null* is similar to CCS's *nil* process. The *null* process is usually used in LCC when an agent needs to carry out internal computational actions (see Figure 9).

5 μ -Calculus: a Modified Version

The previous section introduces LCC which we use in parallel with deontic constraints for modelling MAS scenarios. This constitutes the system model S . We now introduce a modified version of the μ -calculus which is used for specifying properties ϕ . The system model S is then fed to the model checker along with property specifications ϕ to verify whether or not S satisfies ϕ .

The μ -calculus, or the modal μ -calculus as named by [15], extends modal logic by introducing fixed point operators which provide recursion. The modal operators — \Box for *necessarily* and \Diamond for *possibly* — provide the means for expressing the branching logic operators (*for every path* A and *there exists a path* E). Recursion, on the other hand, provides the means for expressing all the usual temporal operators (the *next* X, *until* U, *eventually* F, and *always* G operators) [3]. This results in a simpler syntax yet denser formulae.

To avoid undesirable looping behaviours explained in Section 6.2, our model checker uses the alternation-free μ -calculus [5] — a fragment of our slightly modified version of μ -calculus — where nesting of minimal and maximal fixed-point operators are prohibited [2]. This simplified syntax

(Figure 10), in addition to restricting the language to the alternation-free fragment, is another reason behind the compact model checker (Figure 12) introduced in Section 6. Furthermore, the alternation-free μ -calculus, as discussed by [4] and [9], is expressive enough to subsume most logics and formalisms, such as CTL and ACTL.

5.1 μ -Calculus Syntax and Semantics

The syntax and semantics of the μ -calculus are provided by Figure 10 [16].

For a given set of states P and a valuation function V which maps

The semantics imply that a state E always satisfies **tt**, and never **ff**. The propositional variable Z is satisfied if E belongs to the valuation of Z . E satisfies $\phi_1 \wedge \phi_2$ if it satisfies both ϕ_1 and ϕ_2 , and it satisfies $\phi_1 \vee \phi_2$ if it satisfies either ϕ_1 or ϕ_2 . $\langle A \rangle \phi$ is satisfied if E can take an action a , element of A , to state F , such that F satisfies ϕ . Similarly, $[A]\phi$ is satisfied if for all actions a_i that E can take to F_i , where a_i is an element of A , then F_i satisfies ϕ . $\nu Z.\phi(Z)$ is satisfied if E belongs to the union of all post-fixed points, while $\mu Z.\phi(Z)$ is satisfied if E belongs to the intersection of all pre-fixed points.

The modifications we made to the language is that A , the set of actions, may now contain message passing actions as well as non-communicative actions (or internal computations) $\#(C)$, where C is a constraint to be satisfied by some agent or any other constraint we would like to verify.

Syntax:

$$\phi ::= \mathbf{tt} \mid \mathbf{ff} \mid Z \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \langle A \rangle \phi \mid [A]\phi \mid \nu Z.\phi(Z) \mid \mu Z.\phi(Z)$$

Semantics:

For a given set of states P and a valuation function V which maps propositional variables to elements of 2^P :

$$\begin{aligned} E &\models \mathbf{tt} \\ E &\not\models \mathbf{ff} \\ E &\models Z && \text{iff } E \in V(Z) \\ E &\models \phi_1 \wedge \phi_2 && \text{iff } E \models \phi_1 \text{ and } E \models \phi_2 \\ E &\models \phi_1 \vee \phi_2 && \text{iff } E \models \phi_1 \text{ or } E \models \phi_2 \\ E &\models \langle A \rangle \phi && \text{iff } \exists F \in \{E' : E \xrightarrow{a} E' \text{ and } a \in A\}. F \models \phi \\ E &\models [A]\phi && \text{iff } \forall F \in \{E' : E \xrightarrow{a} E' \text{ and } a \in A\}. F \models \phi \\ E &\models \nu Z.\phi(Z) && \text{iff } E \in \bigcup \{S : S \subseteq \|\phi(Z)\|\} \\ E &\models \mu Z.\phi(Z) && \text{iff } E \in \bigcap \{S : \|\phi(Z)\| \subseteq S\} \end{aligned}$$

tt and **ff** are the logical *true* and *false*, respectively.

Z is a propositional variable.

A is a set of actions

E and F are states of the transition system.

S is a subset of P .

$V(Z)$ is the set of states satisfying Z with respect to the valuation V .

$\|\phi(Z)\|$ is the set of states satisfying $\phi(Z)$.

Figure 10: μ -Calculus syntax and semantics

5.2 μ -Calculus for Specifying MAS Properties

Although selecting a suitable description language is necessary for deciding what aspects of an interaction may be modelled by the specification, selecting the right temporal logic is only as important. It is the temporal logic which controls what behavioural aspects of the system model may be verified. In what follows, we give an idea of the type of properties that may be verified using our modified μ -calculus version.

Let us consider the travel agency scenario of Figure 7. The broker needs to verify that certain properties, required by the agents, are satisfied. For example, the customer is interested in an interaction that guarantees providing flight and hotel quotations. Property 1 verifies the message passing actions of an interaction protocol. It states that if the customer agent requests a vacation package (by sending the *vacation_details* message), then the travel agent will always eventually send back a list of flights and hotels (by sending the *available_flights* and *available_hotels* messages).

$$\begin{aligned} \nu Z. [-]Z \wedge [out(vacation_details(_, _, _), a(travel_agent(_, _, _), _))] \\ ((\mu Y. \langle - \rangle \mathbf{tt} \wedge [in(available_flights(_), a(travel_agent(_, _, _), _))]Y) \wedge \\ (\mu X. \langle - \rangle \mathbf{tt} \wedge [in(available_hotels(_), a(travel_agent(_, _, _), _))]X)) \end{aligned} \quad (1)$$

Property 1 is read as follows: It is always the case ($\nu Z. [-]Z$) that if a request for a vacation package is made ($out(vacation_details(_, _, _), a(travel_agent(_, _, _), _))$) then eventually ($\mu Y. \langle - \rangle \mathbf{tt}$) a list of available flights and hotels will be received ($in(available_flights(_), a(travel_agent(_, _, _), _))$) and $in(available_hotels(_), a(travel_agent(_, _, _), _))$).

While property 1 above verifies the correctness of the message passing actions, our modified version of the μ -calculus allows us to test for non-communicative actions as well. This gives way

to verifying deontic rules in parallel with interaction rules. For example, the broker will need to verify that the travel agent is capable of accessing the hotel directory. Property 2 models this: In every run of the interaction, it is always the case that the hotel directory is eventually queried. The property is said to be satisfied if the LCC constraint $query(HD, D, HL)$ on the *travel_agent* role of Figure 7 can be satisfied. However, this property holds only if the deontic rules of the travel agent and the hotel directory provide the travel agent with the access needed to perform its query (see Rules 1 and 2 of Table 1).

$$\mu Z. \langle \#(query(HD, -, -)) \rangle \mathbf{tt} \vee (\langle - \rangle \mathbf{tt} \wedge [-]Z) \quad (2)$$

6 The Model Checker

Following the introduction of the LCC and the μ -calculus languages used for specifying system models and temporal properties, respectively, this section introduces the model checker’s technique and algorithm. Model checking LCC protocols suggests — given its logic programming nature along with its clause expansion mechanism — the use of a logic based local model checker.

6.1 Local Model Checking

In global model checking, the whole state-space is generated and the states satisfying the property ϕ are computed. On the other hand, local model checking techniques tend to verify whether a single state s_0 satisfies the property ϕ . The search is performed by traversing the state-space starting at s_0 and terminating when a solution is reached. To verify the satisfaction of a property ϕ by a system model M , the model checker is fed the initial state s_0 of M along with the property ϕ .

In LCC, agents’ actions are a result of the transition rules applied to the protocols (Figure 9). With each transition, the agent is traversing the state-space graph (or the transition graph [16]). For example, the transition graph of the process $a(paying_customer(CD, T), C)$ of Figure 7 is provided by Figure 11. The state s_0 , the initial state of this process, can only take a transition $out(payment(PD))$ to state s_1 , where $payment(PD)$ is the message sent and out is the name of the channel between the two communicating agents, the customer and the travel agent. The interaction proceeds, traversing the transition graph one step at a time. Note that when model checking a system, the transition graph of the whole system is used.

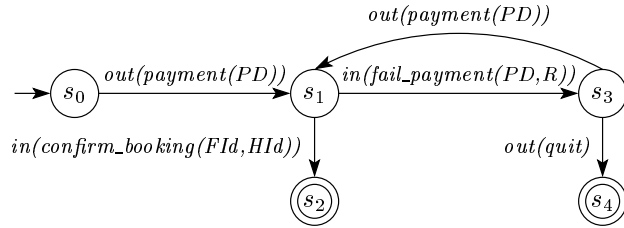


Figure 11: Transition Graph of process $a(paying_customer(CD, T), C)$ of Figure 7

The nature of the LCC language and its transition rules proposes the use of a local model checker that would not construct the complete state graph before its testing for satisfaction, but move through the state space one step at a time until a result is reached. Model checking approaches based on tableaux systems (e.g. [17]) provide such a solution. Termination, in such scenarios, is then to be addressed. We refer to the XMC model checker for inspiration. The XMC system [11] is a model checker built on top of XSB [14], a tabled Prolog system. The concept of caching in tabled Prolog ensures termination, avoids redundant subcomputations, and computes the well-founded model of normal logic programs. We rationally reconstruct the XMC model to accept LCC models and our verified version of μ -calculus. The result is a simplified and remarkably compact model checker that is based solely on the μ -calculus’ proof rules as well as LCC’s transition rules (Figures 12 and 9, respectively). This significantly simplified version does not affect the model checker’s efficiency discussed in Section 8, the base of which is the XSB system.

6.2 Model Checking Algorithm

We now define our model checking algorithm as follows. A system S is said to satisfy a formula ϕ if its initial state E satisfies ϕ . Hence, the initial state E and the formula ϕ are passed to the model checker. Model checking is then performed in a top-down manner based on the μ -calculus

semantics presented in Figure 10. Rules concerning the verification of $E \models \mathbf{tt}$, $E \not\models \mathbf{ff}$, $E \models Z$, $E \models \phi_1 \wedge \phi_2$, $E \models \phi_1 \vee \phi_2$, $E \models \langle A \rangle \phi$ and $E \models [A] \phi$ can easily be encoded in Prolog. The challenge is dealing with the greatest and least fixed point formulae. Prolog, by nature, computes the least fixed point solution. The greatest fixed point, however, is the dual of the least fixed point, i.e. the greatest fixed point formula is satisfied if the least fixed point of the negated formula fails to be satisfied. In XSB, this can be achieved by making use of the negation predicate *sk_not/1* in addition to the tabled environment which ensures that a least fixed point solution is found if it does exist in the table.

The result is a simple, straightforward, and compact XSB coded model checker: the code is directly translated into XSB from Figure 12, where $E \xrightarrow{A} F$ follows LCC's transition rules of Figure 9. However, this simple and straightforward algorithm limits the μ -calculus to the alternation-free fragment. That is because formulae with alternation result in loops through negation which are not easily handled by XSB.

It is worth noting that the model checker does not use the traditional notion of channels. A process calculus usually uses the concept of a channel, through which messages are passed, to connect processes together. This helps in specifying the direction of flow of messages, restricting channels to specific processes, etc. In multi-agent systems, agents should be capable of sending/receiving messages to/from any other agent. Each agent should have channels connecting it to all other agents in the system. Restrictions on who can send messages to who should be dealt with at a higher level by the agents themselves. Instead of using channels to specify where messages are sent, LCC directly specifies the agent (or process) to which the message is sent to. Our model checker implements this by introducing a list of messages M_i . The sending and receiving agent details are attached to the transmitted messages which are saved in M_i . When an agent is expecting a message, it searches for it in the list of incoming messages M_i .

$models(E, \mathbf{tt})$	\leftarrow	\mathbf{true}
$models(E, \phi_1 \vee \phi_2)$	\leftarrow	$models(E, \phi_1) \vee models(E, \phi_2)$
$models(E, \phi_1 \wedge \phi_2)$	\leftarrow	$models(E, \phi_1) \wedge models(E, \phi_2)$
$models(E, \langle A \rangle \phi)$	\leftarrow	$\exists F. ((E \xrightarrow{A} F) \wedge models(F, \phi))$
$models(E, [A] \phi)$	\leftarrow	$\forall F. ((E \xrightarrow{A} F) \rightarrow models(F, \phi))$
$models(E, \mu Z. \phi)$	\leftarrow	$models(E, \phi)$
$models(E, \nu Z. \phi)$	\leftarrow	$dual(\phi, \phi') \wedge \neg models(E, \phi')$

Figure 12: The modal μ -calculus proof rules

7 Related Work

In this field, different verification techniques have been applied to various aspects of multi-agent systems. In [20] and [1], message passing actions affect the mental states of agents (or the beliefs, desires, and intentions). The verification process is carried by testing the mental states of the agents involved in an interaction. Consequently, the verified system's results hold for a particular set of agents. The approach presented in this paper separates the interaction layer from the agents BDI layer (Figure 1). The model checker, however, is dynamic. This allows it to be invoked at run-time to verify properties affected by interaction and/or dynamic deontic rules.

[6] offers a technique which separates the mental state of agents from the social state of an interaction. Both the system specification and the properties to be verified are written in DTL. Verification is carried based on the use of Buchi automata. Model checking is then applied to the proof of the formulas to be verified. This makes the verification process a complex process based on a combination of model checking and other techniques. This paper proposes a simple and automatic technique which allows the agents themselves to perform the verification process at run-time.

[18] and [19] also separate the mental state from the social state. These approaches strictly limit the verification process to verifying message passing actions only. The approach presented in this paper allows the model checker to verify message passing actions of the interaction model as well as deontic constraints affecting the interaction. This is made possible by introducing the modified version of the μ -calculus (Section 5) which permits the verification of constraints in LCC which are essentially a link to the deontic constraints.

[21] differs from all other techniques since it focuses on the evolution of knowledge in MAS. The system is a Real Time Interpreted System. The temporal logic used is TECTLK — a logic for knowledge and real time. All system states should be represented by bit vectors and are encoded before the verification process is initiated. These undergo a translation process and the resulting

propositional formula is fed to a SAT solver for verification. This technique differs from the one presented in this paper since it verifies the change of knowledge in MAS. Furthermore, all system states need to be encoded before the verification process can take place. This paper proposes a technique which requires only the initial state of the system. Local model checking is used to incrementally generate the state-space until a solution is reached. This is applied via the proof rules of Figure 12 and the transition rules of Figure 9. The whole process is an automatic process that could be carried out by agents at run-time.

8 Conclusion

This paper presents a model checker which may be invoked at run-time by agents for verifying instances of the interaction protocol. Each of the languages chosen — LCC, policy languages, or μ -calculus — contribute to the following features of the model checker:

1. **LCC for supporting the attachment of a deontic model to the interaction model**
Policy languages are essentially a tuple of the form $(s, o, < sign > a)$ which permits or prohibits — depending on the *sign* of a — a subject s from executing the action a on an object o . Additionally, conditions may be attached to rules (see Table 1). In short, policy languages are basically a set of constraints. Constraints in LCC could then act as a window to the dynamic, agent specific set of deontic rules. For example, constraint $query(HD, D, HL)$ of the travel agency interaction protocol triggers deontic Rules 1 and 2 of Table 1.
2. **Logic-based programming for efficient constraint computing**
The use of computational logic results in a model checker that deals efficiently with constraints and complex data structures. This is crucial for us since our system model makes heavy use of constraints and structured terms.
3. **A compact size model checker for agents to use at run-time**
The use of the modal μ -calculus along with local model checking techniques, which is suggested by the transition rules of LCC, results in a very simple and compact model checker. The model checker is constructed from the rules of Figures 9 and 12 and encoded in XSB. This throws the burden of searching the state-space on the underlying XSB system. The small size of the model checker makes it a good candidate for the use by agents at run-time.
4. **LCC for supporting dynamic model checking of dynamic system models**
The decentralised coordination mechanism driven by the clause expansion mechanism of LCC allows agents to verify dynamic deontic rules as well as dynamic interaction protocols at run-time. This is made possible by retrieving the current protocol state (at run-time) along with the current deontic rules, and feeding this information to the model checker.

References

- [1] Massimo Benerecetti, Fausto Giunchiglia, and Luciano Serafini. Model checking multiagent systems. *Journal of Logic and Computation*, 8(3):401–423, June 1998.
- [2] Benedikt Bollig, Martin Leucker, and Michael Weber. Local parallel model checking for the alternation-free mu-calculus. In *Proceedings of the 9th International SPIN Workshop on Model checking of Software (SPIN '02)*, volume 2318 of *Lecture Notes in Computer Science*, pages 128–147. Springer-Verlag Inc., 2002.
- [3] Julian Bradfield and Colin Stirling. Modal logics and mu-calculi: an introduction. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra*, pages 293–330. Elsevier, North-Holland, 2001.
- [4] E. Allen Emerson. Model checking and the mu-calculus. In Neil Immerman and Phokion G. Kolaitis, editors, *Descriptive Complexity and Finite Models, Proceedings of DIMACS Workshop*, volume 31 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 185–214. American Mathematical Society, 1996.

- [5] E. Allen Emerson and Chin-Laung Lei. Efficient model checking in fragments of the propositional mu-calculus (extended abstract). In *Proceedings of the First IEEE Symposium on Logic in Computer Science*, pages 267–278, Los Alamitos, CA, 1986. IEEE Computer Society Press.
- [6] Laura Giordano, Alberto Martelli, and Camilla Schwind. Verifying communicating agents by model checking in a temporal action logic. In *Proceedings of Logics in Artificial Intelligence, 9th European Conference, JELIA 2004*, volume 3229 of *Lecture Notes in Computer Science*, pages 57–69, Lisbon, Portugal, 2004. Springer.
- [7] Hao He, Hugo Haas, and David Orchard. Web services architecture usage scenarios. *W3C Working Group Note*, Feb 2003. <http://www.w3.org/TR/2004/NOTE-ws-arch-scenarios-20040211/>.
- [8] Charles Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [9] Radu Mateescu and Mihaela Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. *Science of Computer Programming*, 46(3):255–281, 2003.
- [10] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.
- [11] C. R. Ramakrishnan, I. V. Ramakrishnan, Scott A. Smolka, Yifei Dong, Xiaoqun Du, Abhik Roychoudhury, and V. N. Venkatakrisnan. XMC: a logic-programming-based verification toolset. In *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 576–580. Springer, 2000.
- [12] David Robertson. A lightweight coordination calculus for agent social norms. In *Proceedings of Declarative Agent Languages and Technologies workshop at AAMAS*, New York, USA, 2004.
- [13] David Robertson. Multi-agent coordination as distributed logic programming. In *International Conference on Logic Programming*, volume 3132 of *Lecture Notes in Computer Science*, pages 416–430, Sant-Malo, France, 2004. Springer.
- [14] Konstantinos Sagonas, Terrance Swift, and David S. Warren. XSB as an efficient deductive database engine. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data (SIGMOD '94)*, pages 442–453. ACM Press, 1994.
- [15] Colin Stirling. Modal and temporal logics. In *Handbook of Logic in Computer Science: Background - Computational Structures*, volume 2, pages 477–563. Oxford University Press, Inc., New York, NY, USA, 1992.
- [16] Colin Stirling. *Modal and Temporal Properties of Processes*. Texts in Computer Science. Springer-Verlag, 2001.
- [17] Colin Stirling and David Walker. Local model checking in the modal mu-calculus. *Theoretical Computer Science*, 89(1):161–177, oct 1991.
- [18] Chris Walton. Model checking agent dialogues. In *Proceedings of the Workshop on Declarative Agent Languages and Technologies (DALT '04)*, volume 3476 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.
- [19] Wu Wen and Fumio Mizoguchi. Analysis and verification of multi-agent interaction protocols. In *Proceedings of the Sixth Asia-Pacific Software Engineering Conference (APSEC '99)*, pages 252–259, Takamatsu, Japan, 1999. IEEE Computer Society.
- [20] Michael Wooldridge, Michael Fisher, Marc-Philippe Huget, and Simon Parsons. Model checking multi-agent systems with mable. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems (AAMAS '02)*, pages 952–959, New York, USA, 2002. ACM Press.
- [21] B. Woźna, A. Lomuscio, and W. Penczek. Bounded model checking for knowledge and real time. In *In Proceedings of the 4th International Joint Conference on Autonomous Agents and Multi-agent systems (AAMAS'05)*, pages 165–172. ACM Press, 2005.