OpenKnowledge

FP6-027253

Initial Test Rigs

Paolo Besana, Phil Graham, David Lambert,
Nardine Osman, and David Robertson

School of Informatics, University of Edinburgh, UK

Report Version: final
Report Preparation Date:
Classification: deliverable D7.1
Contract Start Date: 1.1.2006 Duration: 36 months
Project Co-ordinator: University of Edinburgh (David Robertson)
Partners: IIIA(CSIC) Barcelona
Vrije Universiteit Amsterdam
University of Edinburgh
KMI, Open University
University of Southampton
University of Trento

Protocol Based Analysis for Distributed Knowledge Sharing

Paolo Besana, Phil Graham, David Lambert, Nardine Osman, David Robertson

Informatics,
University of Edinburgh,
UK

Abstract

Formal verification and analysis has made few inroads into knowledge sharing because, at the level of shared knowledge, the computation used to perform sharing normally is too complex, unpredictable and/or obscure to allow practical methods to be developed. Recently, however, computational methods have been developed that apply directly for knowledge sharing some of the methods that have traditionally been used in formal verification. The area to which these methods is applied, however, is very different in nature to those of traditional formal verification. This leads us to apply formal analytical methods in novel ways. We describe four of these.

When verification methods are applied to distributed systems the focus of analysis normally is on generic system properties (such as liveness or deadlock). When we are dealing with low-level components, such as basic communication protocols, then these generic properties are (arguably) the only properties of immediate concern. When we tackle knowledge sharing in distributed systems then the range of possible properties to analyse is richer, and the methods of analysis are more varied. This paper describes four categories of analysis within a single framework for distributed knowledge sharing. Our framework uses the Lightweight Coordination Calculus (LCC) to specify the knowledge sharing interaction. LCC is an executable specification language and is being used in distributed knowledge sharing applications (most notably as part of the OpenKnowledge project, `www.openk.org`).

The paper is structured as follows. First, in Section 1 we describe a top-level framework for computation using LCC. Then, in sections 2 to 5 we identify four properties of knowledge sharing in this framework and for each show a form of analysis appropriate to it. All of the properties we consider are, in the general case, impossible to guarantee in a distributed system so the aim of analysis is not to conclusively guarantee our properties but to make it more likely that are preserved when we engineer systems.

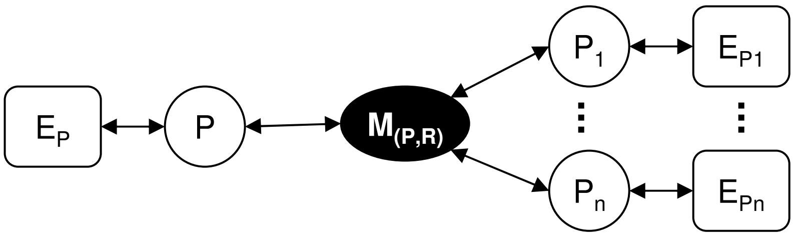
1 A Model of Computation for Peer to Peer Knowledge Sharing

The model of computation that we use as the basis for our framework is that employed by the OpenKnowledge project, using the Lightweight Coordination Calculus (LCC). This is described in detail in [9] but we repeat the relevant part of that description in this section and connect it to the framework summary (in Definition 1) that we use as a recurrent pattern when we get to the analytical methods of Sections 2 to 5.

We assume that distributed knowledge sharing is performed by some collection of computational processes (*e.g* Web services, peers in a peer-to-peer network, or agents in a multi-agent system). We refer to such processes as peers, not because peer-to-peer infrastructure essential to us but because this emphasises the isolated and autonomous nature of the processes. Each peer may interact, privately, with its local environment. When a peer wishes to interact with other peers then this is achieved through an explicit model of interaction (defined in LCC, see below) that constrains the sequence of message passing between peers. To participate in an interaction model a peer must adopt one or more roles in that model - hence peers become involved in interactions by subscribing to roles defined for that interaction. In [9] we show the correspondences between this model of interaction and those of other commonly used coordination systems.

In all cases there is a correspondence at the level of specification - so LCC can be used to specify many of the forms of interaction found in those systems. In addition, however, we have produced a mechanism allowing direct deployment of LCC, so it can be used as an executable specification language.

Definition 1 *In a peer to peer knowledge sharing system a peer is a computational process, P, with a local environment, E_P. When in an interaction with other peers, a peer adopts some role, R, that conforms to an explicit model of the interaction, $M_{(P,R)}$. This is depicted in the diagram below*



where:

P	$=$	*process name*
R_P	$=$	*role of* P
$M_{(P,R)}$	$=$	*interaction model for* R_P
E_P	$=$	*environment of* P
$\mathcal{R}_P$	$=$	$\{R_P, \ldots\}$
$\mathcal{M}_P$	$=$	$\{M_{(P,R)}, \ldots\}$
$\mathcal{P}(M_{(P,R)})$	$=$	$\{P_i, \ldots\}$ *such that* P_i *participates in* $M_{(P,R)}$

Figure 1 defines the syntax of LCC. An interaction model in LCC is a set of clauses, each of which defines how a role in the interaction must be performed. Roles are described by the type of role and an identifier for the individual peer undertaking that role. The definition of performance of a role is constructed using combinations of the sequence operator ('*then*') or choice operator ('*or*') to connect messages and changes of role. Messages are either outgoing to another peer in a given role ('$\Rightarrow$') or incoming from another peer in a given role ('$\Leftarrow$'). Message input/output or change of role can be governed by a constraint defined using the normal logical operators for conjunction, disjunction and negation. Notice that there is no commitment to the system of logic through which constraints are solved - so different peers might operate different constraint solvers (including human intervention).

In [9] a full formal description of a computation method using LCC is described. For our current purposes, however, it is sufficient to repeat only the transition rules used by each peer to advance the state of its role in the interaction. This is done by selecting the appropriate clause, $\mathcal{S}_p$, for that role and we now explain how to advance the state associated with this role to the new version of that clause, $\mathcal{S}'_p$, given an input message set, M_i, and producing a new message set, M_n, which contains those messages from M_i that have not been processed plus additional messages added by the state transition. Since we shall need a sequence of transitions to the clause for $\mathcal{S}_p$ we use C_i to denote the start of that sequence and C_j the end. The rewrite rules of Figure 2 are applied to give the transition sequence of expression 1.

$$
\begin{array}{rcl}
Model & := & \{Clause, \ldots\} \\
Clause & := & Role :: Def \\
Role & := & a(Type, Id) \\
Def & := & Role \mid Message \mid Def \textit{ then } Def \mid Def \textit{ or } Def \\
Message & := & M \Rightarrow Role \mid M \Rightarrow Role \leftarrow C \mid M \Leftarrow Role \mid C \leftarrow M \Leftarrow Role \\
C & := & Constant \mid P(Term, \ldots) \mid \neg C \mid C \wedge C \mid C \vee C \\
Type & := & Term \\
Id & := & Constant \mid Variable \\
M & := & Term \\
Term & := & Constant \mid Variable \mid P(Term, \ldots) \\
Constant & := & \text{lower case character sequence or number} \\
Variable & := & \text{upper case character sequence or number}
\end{array}
$$

Figure 1: LCC syntax

$$
\begin{array}{rcl}
C_i \xrightarrow{M_i, \mathcal{S}, M_n} C_j & \leftrightarrow & \exists R, D.(C_i = a(R,p) :: D) \wedge \\
& & \left(\begin{array}{ccc}
C_i & \xrightarrow{R_i, M_i, M_{i+1}, \mathcal{S}, O_i} & C_{i+1} \wedge \\
C_{i+1} & \xrightarrow{R_i, M_{i+1}, M_{i+2}, \mathcal{S}, O_{i+1}} & C_{i+2} \wedge \\
\ldots & & \\
C_{j-1} & \xrightarrow{R_i, M_{j-1}, M_j, \mathcal{S}, O_j} & C_j
\end{array} \right) \wedge \\
& & M_n = M_j \cup O_j
\end{array} \quad (1)
$$

2 Ensuring that Interactions are Available

Using LCC we make interaction models explicit, so $M_{(P,R)}$ in Definition 1 can be communicated between peers. There are, however, potentially a huge number of interactions in which a peer might wish to be engaged, and this makes it a major issue for a peer to know exactly what sort of interaction to attempt with its peers. There are two closely related aspects to this problem: the discovery of appropriate models of interaction and the choice of appropriate peers with which to engage in interactions that have been discovered.

Property 1 Interaction availability: *If a role, R, is in the set of roles, $\mathcal{R}_P$, that a peer, P, wishes to undertake then at some future time an interaction model, $M_{(P,R)}$, should exist in the set of models, $\mathcal{M}_P$, known to P and that model should be capable of being initiated by P on role R and then satisfied via per-to-peer interaction.*

$$R \in \mathcal{R}_P \quad \rightarrow \quad \diamond(\exists M_{(P,R)}.M_{(P,R)} \in \mathcal{M}_P \ \wedge \ (i(M_{(P,R)}) \ \rightarrow \ \diamond a(M_{(P,R)})))$$

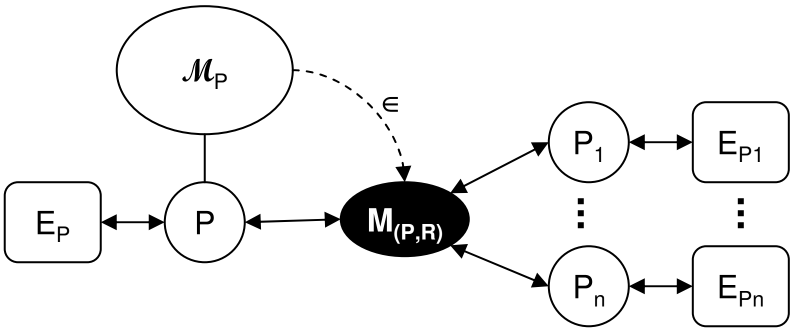
$$
\begin{array}{lll}
R :: B \xrightarrow{R_i, M_i, M_o, \mathcal{S}, O} A :: E & if & B \xrightarrow{R, M_i, M_o, \mathcal{S}, O} E \\
A_1 \ or \ A_2 \xrightarrow{R_i, M_i, M_o, \mathcal{S}, O} E & if & \neg closed(A_2) \wedge \\
 & & A_1 \xrightarrow{R_i, M_i, M_o, \mathcal{S}, O} E \\
A_1 \ or \ A_2 \xrightarrow{R_i, M_i, M_o, \mathcal{S}, O} E & if & \neg closed(A_1) \wedge \\
 & & A_2 \xrightarrow{R_i, M_i, M_o, \mathcal{S}, O} E \\
A_1 \ then \ A_2 \xrightarrow{R_i, M_i, M_o, \mathcal{S}, O} E \ then \ A_2 & if & A_1 \xrightarrow{R_i, M_i, M_o, \mathcal{S}, O} E \\
A_1 \ then \ A_2 \xrightarrow{R_i, M_i, M_o, \mathcal{S}, O} A_1 \ then \ E & if & closed(A_1) \wedge \\
 & & A_2 \xrightarrow{R_i, M_i, M_o, \mathcal{S}, O} E \\
C \leftarrow M \Leftarrow A \xrightarrow{R_i, M_i, M_i - \{m(R_i, M \Leftarrow A)\}, \mathcal{S}, \emptyset} c(M \Leftarrow A) & if & m(R_i, M \Leftarrow A) \in M_i \wedge \\
 & & satisfy(C) \\
M \Rightarrow A \leftarrow C \xrightarrow{R_i, M_i, M_o, \mathcal{S}, \{m(R_i, M \Rightarrow A)\}} c(M \Rightarrow A) & if & satisfied(\mathcal{S}, C) \\
null \leftarrow C \xrightarrow{R_i, M_i, M_o, \mathcal{S}, \emptyset} c(null) & if & satisfied(\mathcal{S}, C) \\
a(R, I) \leftarrow C \xrightarrow{R_i, M_i, M_o, \mathcal{S}, \emptyset} a(R, I) :: B & if & clause(\mathcal{S}, a(R, I) :: B) \wedge \\
 & & satisfied(\mathcal{S}, C)
\end{array}
$$

An interaction model term is decided to be closed as follows:

$$
\begin{array}{l}
closed(c(X)) \\
closed(A \ then \ B) \leftarrow closed(A) \wedge closed(B) \\
closed(X :: D) \leftarrow closed(D)
\end{array}
\qquad (2)
$$

$satisfied(\mathcal{S}, C)$ is true if constraint C is satisfiable given the peer's current state of knowledge. $clause(\mathcal{S}, X)$ is true if clause X appears in the interaction model $\mathcal{S}$, as defined in Figure 1.

Figure 2: Rewrite rules for expansion of an interaction model clause



In general, there is no guarantee that a peer will discover the right interaction model because this involves, in the worst case, searching the entire peer network which is not closed or synchronised. There is also no general guarantee that an interaction model, once initiated will result in a successful outcome for the peer

Figure 3: Rewrite rules governing matchmaking for an LCC protocol

These rewrite rules constitute an extension to those described in Figure 2. A rewrite rule

$$\alpha \xrightarrow{M_i, M_o, \mathcal{P}, O, \mathcal{C}, \mathcal{C}'} \beta$$

holds if α can be rewritten to β where: M_i are the available messages before rewriting; M_o are the messages available after the rewrite; $\mathcal{P}$ is the protocol; O is the message produced by the rewrite (if any); $\mathcal{C}$ is set of collaborators before the rewrite; and $\mathcal{C}'$ (if present) is the—possibly extended—set of collaborators after the rewrite. $\mathcal{C}$ is a set of pairs of role and service name, e.g. *col*(*black-hole-finder*, *ucsd-sdsc*)}. The same rewrite rules hold regardless of the implementation of the matchmaking function *recruit*. This enables us to apply other LCC tools, such as model-checkers and the interpreter itself, without alteration while allowing us to change *recruit*, and means clients can use their own choice of matchmaker and matchmaking scheme.

$$\begin{array}{lll}
A :: B \xrightarrow{M_i, M_o, \mathcal{P}, \mathcal{C}, O} A :: E & if & B \xrightarrow{M_i, M_o, \mathcal{P}, \mathcal{C}, O} E \\
A_1 \ or \ A_2 \xrightarrow{M_i, M_o, \mathcal{P}, \mathcal{C}, O} E & if & \neg closed(A_2) \wedge A_1 \xrightarrow{M_i, M_o, \mathcal{P}, \mathcal{C}, O} E \\
A_1 \ or \ A_2 \xrightarrow{M_i, M_o, \mathcal{P}, \mathcal{C}, O} E & if & \neg closed(A_1) \wedge A_2 \xrightarrow{M_i, M_o, \mathcal{P}, \mathcal{C}, O} E \\
A_1 \ then \ A_2 \xrightarrow{M_i, M_o, \mathcal{P}, \mathcal{C}, O} E \ then \ A_2 & if & A_1 \xrightarrow{M_i, M_o, \mathcal{P}, \mathcal{C}, O} E \\
A_1 \ then \ A_2 \xrightarrow{M_i, M_o, \mathcal{P}, \mathcal{C}, O} A_1 \ then \ E & if & closed(A_1) \wedge A_2 \xrightarrow{M_i, M_o, \mathcal{P}, \mathcal{C}', O} E \\
 & & \wedge collaborators(A_1) = \mathcal{C}' \\
C \leftarrow M \Leftarrow A \xrightarrow{M_i, M_i \setminus \{M \Leftarrow A\}, \mathcal{P}, \mathcal{C}, \emptyset} c(M \Leftarrow A)\mathcal{C} & if & (M \Leftarrow A) \in M_i \wedge satisfied(C) \\
M \Rightarrow A \leftarrow C \xrightarrow{M_i, M_i, \mathcal{P}, \mathcal{C}, \mathcal{C}', \{M \Rightarrow A\}} c(M \Rightarrow A)\mathcal{C}' & if & satisfied(C) \wedge \\
 & & \mathcal{C}' = recruit(\mathcal{P}, \mathcal{C}, role(A)) \\
null \leftarrow C \xrightarrow{M_i, M_i, \mathcal{P}, \mathcal{C}, \emptyset} c(null)\mathcal{C} & if & satisfied(C) \\
a(R, I) \leftarrow C \xrightarrow{M_i, M_o, \mathcal{P}, \mathcal{C}, \emptyset} a(R, I) :: B & if & clause(\mathcal{P}, C, a(R, I) :: B) \\
 & & \wedge satisfied(C)
\end{array}$$

$$\begin{array}{lcl}
collaborators(c(Term)\mathcal{C}) & = & \mathcal{C} \\
collaborators(A_1 \ then \ A_2) & = & collaborators(A_1) \cup collaborators(A_2) \\
collaborators(A :: B) & = & collaborators(A) \cup collaborators(B)
\end{array}$$

because the other peers that engage in the interaction may not behave reliably, even if they did so on earlier occasions. For these reasons, attention has focused on statistically based methods of matchmaking, so that previous knowledge of successes and failures in interacting can be used to inform choices of peers to recruit for interaction models.

In order to simulate the process of matchmaking for an LCC interaction model ($M_{(P,R)}$ in Property 1) we extend the rewrite rules of Figure 2 to include a matchmaking function *recruit*. This function takes as input a set of potential collaborators appropriate to the current role in the interaction model. The rules governing execution of a protocol are in figure 3.

In Section 2.2 we discuss the issue of selecting agents. Many different methods can be used for this. We have chosen a probabilistic method, based on the incidence calculus [2]. This is a truth-functional probabilistic calculus in which the probabilities of composite formulae are computed from intersections and unions of the sets of worlds for which the atomic formulae hold true, rather than from the numerical values of the probabilities of their components. The probabilities are then derived from these incidences. Crucially, in general $P(\phi \wedge \psi) \neq P(\phi) \cdot P(\psi)$. This fidelity is not possible in normal probabilistic logics, where probabilities of composite formulae are derived only from the probabilities of their component formulae. In the incidence calculus, we return to the underlying sets of incidences, giving us more accurate values for compound probabilities.

$$\begin{array}{llll} i(\top) & = worlds & i(\bot) & = \{\} \\ i(\alpha \wedge \beta) & = i(\alpha) \cap i(\beta) & i(\alpha \vee \beta) & = i(\alpha) \cup i(\beta) \\ i(\neg\alpha) & = i(\top) \backslash i(\alpha) & i(\alpha \rightarrow \beta) & = i(\neg\alpha \vee \beta) = (worlds \backslash i(\alpha)) \cup i(\beta) \\ P(\phi) & = \frac{|i(\phi)|}{|i(\top)|} & P(\phi|\psi) & = \frac{|i(\phi \wedge \psi)|}{|i(\psi)|} \end{array}$$

The incidence calculus is not frequently applied, since one requires exact incident records to use it. Fortunately, that's exactly what the matchmaker has on hand.

2.1 The Matchmaker

First, we will explain the overall process of executing an LCC protocol, and the matchmaker's place in it. A client has a task or goal it wishes to achieve: using either a pre-agreed look-up mechanism, or by reasoning about the protocols available, the client will select a protocol, with possibly more than one being suitable. This done, it can begin interpreting the protocol, dispatching messages to other agents as the protocol directs. When the protocol requires a message to be sent to an agent that is not yet identified, the sender queries a matchmaker to discover services capable of filling the role. These new agents we term 'collaborators'. The matchmaker selects the service that maximises the probability of a successful outcome given the current protocol type and role instantiations. The protocol is then updated to reflect the agent's selection, and the term $col(Role, Agent)$, instantiated to the requested role and newly chosen agent, is stored in the protocol's common knowledge where it is visible to the participants and the matchmaker.

The success of a protocol and the particular team of collaborators is decided by the client: on completion or failure of a protocol, the client informs the matchmaker whether the outcome was satisfactory to the client. Each completed brokering session is recorded as an incident, represented by an integer. Our propositions are ground predicate calculus expressions. Each proposition has an associated list of worlds (incidents) for which it is true. Initially, the incident database is empty, and the broker selects services at random. As more data is collected, a threshold is reached, at which point the matchmaker begins to use the probabilities.

2.2 Selecting Agents

We explain the selection of agents using a scenario from astrophysics. In our scenario, an astronomer is using the Grid to examine a black hole. Having obtained the LCC protocol in figure 4, she instantiates the *File* variable to the file she wants to work with, and runs the protocol. The protocol is sent first to a *black-hole-finder* service. This service, in turn, requires an *astronomy-database* to provide the file. If a black hole is found the *black-hole-finder* service will pass the data to a visualisation service. Finally, the client will receive a visualisation or notification of failure. Where is the inter-service variation? Consider that the astronomical data file is very large, and thus network bandwidth between sites will be a crucial factor in determining user satisfaction. Thus, some pairs of database and computation centre will outperform other pairs, even though the individuals in each pairing might be equally capable. Indeed, the 'best' database and compute centre may have a dreadful combined score because their network interconnection is weak.

If we imagine how the matchmaker's incidence database would look after several executions of this workflow (most likely by different clients), we might see something like this:

$$\begin{array}{l}
i(protocol(\textsc{black-hole-search}), [1, 2, \ldots, 25]) \\
i(outcome(good), [1, 2, 3, 4, 6, 10, 11, 12, 16, 22, 23, 24]) \\
i(col(astronomy\text{-}database, greenwich), [18, 19, 20, 21, 22, 23, 24, 25]) \\
i(col(astronomy\text{-}database, herschel), [10, 11, 12, 13, 14, 15, 16, 17]) \\
i(col(astronomy\text{-}database, keck), [1, 2, 3, 4, 5, 6, 7, 8, 9]) \\
i(col(black\text{-}hole\text{-}finder, barcelona\text{-}sc), [8, 9, 16, 17, 24, 25]) \\
i(col(black\text{-}hole\text{-}finder, ucsd\text{-}sdsc), [1, 2, 3, 4, 10, 11, 12, 13, 18, 19, 20]) \\
i(col(black\text{-}hole\text{-}finder, uk\text{-}hpcx), [5, 6, 7, 14, 15, 21, 22, 23]) \\
i(col(visualiser, ncsa), [1, 2, \ldots, 25])
\end{array}$$

Each $i(proposition, incidents)$ records the incidents (that is, protocol interactions or executions) in which the proposition is true. We can see that the BLACK-HOLE-SEARCH protocol has been invoked 25 times, and that it has been successful in those incidents where *outcome*(*good*) is true. Further, by intersecting various incidences, we can compute the success of different teams of agents, and obtain predictions for future behaviour. Let us examine the performance of the Barcelona supercomputer:

$$\begin{array}{lcl}
i(col(black\text{-}hole\text{-}finder, barcelona\text{-}sc) \wedge outcome(good)) & = & \{16\} \\
P(outcome(good)|col(black\text{-}hole\text{-}finder, barcelona\text{-}sc)) & = & \frac{|\{16\}|}{|\{8,9,16,17,24,25\}|}
\end{array}$$

This performance is substantially worse than that of the other supercomputers on this task not because *barcelona-sc* is a worse supercomputer than *ucsd-sdsc* or *uk-hpcx*, but because its network connections to the databases required for this task present a bottleneck, reducing client satisfaction.

From this database, the matchmaker can then determine, for a requester, which services are most likely to lead to a successful outcome, given the current protocol and services already selected. That is, the matchmaker tries to optimise

$$argmax_s P(outcome(good)|\mathcal{P}, col(r, s) \cup \mathcal{C})$$

Where $\mathcal{P}$ is the protocol, $\mathcal{C}$ is the current set of collaborators, r is the role requiring a new service selection, and s is the service we are to select.

We have developed two algorithms for choosing services, although others are possible. The first, called RECRUIT-JOINT, fills all the vacancies in a protocol at the outset. It works by computing the joint distribution for all possible permutations of services in their respective roles, selecting the grouping with the largest probability of a good outcome.

The second approach, RECRUIT-INCREMENTAL, is to select only one service at a time, as required by the executing protocol. The various services already engaged in the protocol, on needing to send a message to an as-yet-unidentified service, will ask the matchmaker to find an service to fulfil the role at hand. RECRUIT-INCREMENTAL computes the probability of a successful outcome for each service available for role R given $\mathcal{C}$ ($\mathcal{C}$ being the collaborators chosen so far), and selects the most successful service. To illustrate RECRUIT-INCREMENTAL, imagine the workflow scenario. At first, the astronomer must ask the matchmaker to fill the *black-hole-finder* role. The *BHF* service's first action is to request the data file from an astronomy database. It therefore returns the protocol to the matchmaker, which selects the *astronomy-database* most likely to produce success, given that the *black-hole-finder* is already instantiated to *BHF*.

Both algorithms support the pre-selection of services for particular roles. An example of this might be a client booking a holiday: if it were accumulating frequent flier miles with a particular airline, it could specify that airline be used, and the matchmaker would work around this choice, selecting the best agents given that the airline is fixed. This mechanism also allows us to direct the matchmaker's search: selecting a particular service can suggest that the client wants similar services, from the same social pool, for the other roles, e.g. in a peer-to-peer search, by selecting an service you suspect will be helpful in a particular enquiry, the broker can find further services that are closely 'socially' related to that first one.

Figure 4: LCC dialogue framework for astronomy workflow scenario

```
a(astronomer(File), Astronomer) ::
        search(File) ⇒ a(black-hole-finder, BHF) then
        ( success ⇐ a(black-hole-finder, BHF) then                                                  )
        ( receive-visualisation(Thing, V) ← visualising(Thing) ⇐ a(visualiser, V)                 ) or
        failed ⇐ a(black-hole-finder, BHF)

a(black-hole-finder, BHF) ::
        search(File) ⇐ a(astronomer(File), Astronomer) then
        grid-ftp-get(File) ⇒ a(astronomy-database, AD) then
        ( grid-ftp-sent(File) ⇐ a(astronomy-database, AD) then      )
        ( success ⇒ a(astronomer, Astronomer)                         )
        (           ← black-hole-present(File, Black-hole) then       ) or
        ( visualise(Black-hole, Astronomer) ⇒ a(visualiser, V)        )
        failed ⇒ a(astronomer(File), Astronomer)

a(astronomy-database, AD) ::
        grid-ftp-get(File) ⇐ a(black-hole-finder, BHF)
        grid-ftp-sent(File) ⇒ a(black-hole-finder, BHF) ← grid-ftp-completed(File, AD)

a(visualiser, V) ::
        visualise(Thing, Client) ⇐ a(_, Requester) then
        visualising(Thing) ⇒ a(_, Client) ← serve-visualisation(Thing, Client)
```

Note that LCC is being used only to coordinate the interaction: where appropriate, individual agents may use domain-specific protocols, such as Grid FTP, to perform the heavy lifting or invoke specific services outside of the LCC formalism and communication channel.

We can see from figure 6(a) that using this technique can substantially improve performance over random selection of agents which can individual meet the requirements. Which algorithm should one choose? In protocols where most roles are eventually filled, RECRUIT-JOINT will outperform RECRUIT-INCREMENTAL, since it is not limited by the possibly suboptimal decisions made earlier. RECRUIT-JOINT is also preferable when one wishes to avoid multiple calls to the matchmaker, either because of privacy concerns, or for reasons of communication efficiency. However, in protocols which rarely have all their roles instantiated, RECRUIT-JOINT can end up unfairly penalising those services which have not actually participated in the protocols they are allocated to. RECRUIT-INCREMENTAL is therefore more suitable in protocols where many roles go unfilled: total work on the broker would be reduced, and the results would probably be at least as good as for brokering all services.

2.3 Selecting Roles

So far, we have considered the case where the protocol is defined, and we simply need to select agents to fill the roles. What if the roles themselves are undefined, if the protocol is incompletely specified? What if we allowed agents to begin executing incomplete protocols? If the matchmaker could elaborate protocols at run time, selecting the elaboration based on prior experience? We will show one way to do this using the incidence calculus, in a very similar fashion to how we selected agents.

Roles consist of an ordering of messages, together with constraints, and moves to other roles. It might be the case that just changing the ordering might make a large difference. For instance, if one is arranging to travel to a concert, it is preferable to obtain event tickets first, then organise transport. In our example,

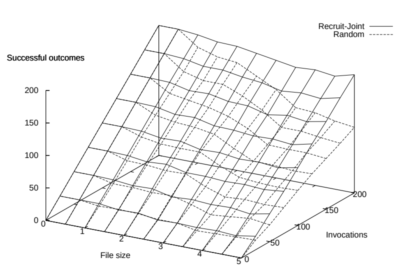
Figure 5: Algorithms

```
RECRUIT-JOINT($protocol$, $database$)
1  $roles \leftarrow$ ROLES-REQUIRED($protocol$)
2  $collaborations \leftarrow$ ALL-COLLABORATIONS($protocol$, $database$, $roles$)
3  for $c \in collaborations$
4      do $quality[c] \leftarrow$ PROBABILITY-GOOD-OUTCOME($protocol$, $database$, $c$)
5  return ARGMAX($collaborations$, $quality$)

RECRUIT-INCREMENTAL($protocol$, $database$, $role$)
1  for $r \in$ ACTIVE-ROLES($protocol$)
2      do $collaborators[r] \leftarrow$ COLLABORATOR-FOR-ROLE($protocol$, $r$)
3  $candidates \leftarrow$ CAPABLE-AGENTS($database$, $role$)
4  for $c \in candidates$
5      do $collaborators[role] \leftarrow c$
6         $quality[a] \leftarrow$ PROBABILITY-GOOD-OUTCOME($database$, $collaborators$)
7  return ARGMAX($candidates$, $quality$)

EMBELLISH-INCREMENTAL($protocol$, $database$, $role$)
1  for $r \in$ ROLE-DEFINITIONS($protocol$)
2      do $role\text{-}definition[r] \leftarrow$ DEFINITION-FOR-ROLE($protocol$, $r$)
3  $candidates \leftarrow$ AVAILABLE-ROLE-DEFINITIONS($protocol$,$database$,$role$) $role$)
4  for $c \in candidates$
5      do $role\text{-}definition[role] \leftarrow c$
6         $quality[c] \leftarrow$ PROBABILITY-GOOD-OUTCOME($database$, $role\text{-}definitions$)
7  return ARGMAX($role\text{-}definitions$, $quality$)
```

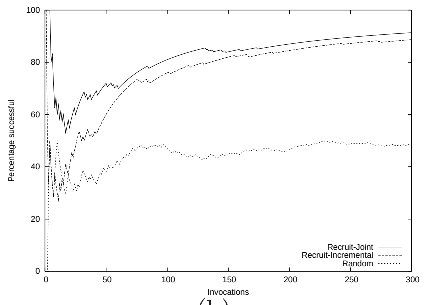
ARGMAX as used here does not always select the highest value. To improve the exploration of options, those entries that have low numbers of data points (i.e. have not often been selected previously) are preferentially chosen, and in other cases a random selection is sometimes made.

we take to problem of booking a trip involving a flight and hotel room. The LCC protocol is shown in figure 7. If we suppose that it is a preferable course of action to book the flight then the hotel room, since hotel room costs are more flexible that flight ones, we can expect a better outcome using *flight-then-hotel* rather than *hotel-then-flight*. Figure 6(c) shows the improvement in a simulation. The algorithm used is EMBELLISH-INCREMENTAL, shown in figure 5. EMBELLISH-INCREMENTAL works similarly to RECRUIT-INCREMENTAL, adding role definitions to the protocol as those roles are required at run-time. We have not provided equivalent to RECRUIT-JOINT, since this can inflate protocols with many roles that will remain unused.

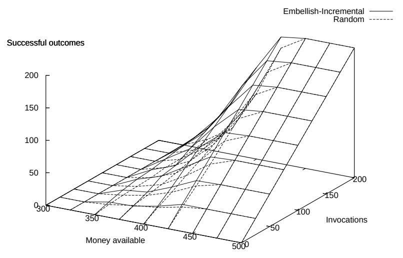
Figure 6: Simulation results



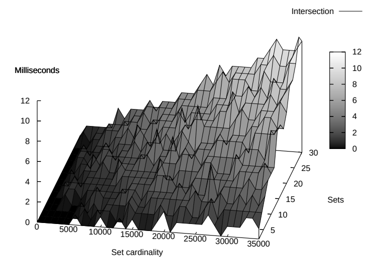
(a)



(b)



(c)



(d)

In (a), we see the improvement in task achievement using agent selection obtained using RECRUIT-JOINT versus random selection. Using the same scenario, but fixing the file size at 5 Gigabytes, (b) shows the relative performance of RECRUIT-JOINT, RECRUIT-INCREMENTAL, and random selection. We can see similar gains for selection of roles in (c), using the travel agent scenario. Performance of the underlying set calculus intersection operation is shown in (d).

2.4 Discussion

Performance seems quite reasonable for very large sets. The core operation of this technique is set intersection, since for every collaboration or set of role definitions, the intersection of their incidences must be computed. By using a heap-sort like intersection algorithm, this can be done in order $O(n \log n)$. Figure 6(d) shows that we can quickly calculate intersections over large sets for reasonable numbers of sizable sets.

We note here two significant problems that seem to be inescapable issues intrinsic to the problem: trusting clients to evaluate protocol performance honestly and in a conventional manner; and the problems of locating mutually cooperative services in a large agent ecology. Since individual client services are responsible for the assigning of success metrics to matchmakings, there is scope for services with unusual criteria or malicious intent to corrupt the database. The second question, largely unasked, is about the likely demographics of service provision. For some types of service, like search, we have already seen that a very small number of providers. For other tasks, a few hundred exist: think of airlines. For some, though, we may millions of

Figure 7: Booking a holiday with LCC

```
a(traveller, Traveller) ::
        book-holiday(Src, Dst, Start, End, Money) ⇒ a(travel-agent, Agent)
            ← travel-details(Src, Dst, Start, End, Money) then
        ( booking(Start, End, Cost) ⇐ a(travel-agent, Agent) then
          matchmaking(good) ⇒ a(matchmaker, matchmaker)            ) or
        ( failure ⇐ a(travel-agent, Agent) then
          matchmaking(bad) ⇒ a(matchmaker, matchmaker)             )
```

Note that the *travel-agent* role is not specified in the client's protocol! We leave it to the matchmaker to find one. The matchmaker, let us say, has the following role definitions available to it:

```
role(flight-then-hotel) ≡ a(travel-agent, Agent) ::
        book-holiday(Src, Dst, Start, End, Money) ⇐ a(client, Client) then
        book-flight(Src, Dst, Start, End, Money) ⇒ a(airline, Airline) then
        ( no-flights ⇐ a(airline, Airline) then
          failure ⇒ a(client, Client)                    ) or
        ( flight-booking(Flight-Cost) ⇐ a(airline, Airline) then
          flight-available(Src, Dst, Start, End, Money) ⇐ a(airline, Airline) then
          book-hotel(Location, Start, End, Money) ⇒ a(hotel, Hotel)
              ← is(Money-Left, Money − Flight-Cost) then
          ( hotel-booking(Hotel-Cost) ⇐ a(hotel, Hotel) then
            booking(Total-Cost) ⇒ a(client, Client)
                ← is(Total-Cost, Flight-Cost + Hotel-Cost)   ) or
          ( no-vacancy ⇐ a(hotel, Hotel) then
            failure ⇒ a(client, Client)                      )
        )

role(flight-then-hotel) ≡ a(travel-agent, Agent) ::
        book-holiday(Src, Dst, Start, End, Money) ⇐ a(client, Client) then
        book-flight(Src, Dst, Start, End, Money) ⇒ a(airline, Airline) then
        ( no-flights ⇐ a(airline, Airline) then
          failure ⇒ a(client, Client)                    ) or
        ( flight-booking(Flight-Cost) ⇐ a(airline, Airline) then
          flight-available(Src, Dst, Start, End, Money) ⇐ a(airline, Airline) then
          book-hotel(Location, Start, End, Money) ⇒ a(hotel, Hotel)
              ← is(Money-Left, Money − Flight-Cost) then
          ( hotel-booking(Hotel-Cost) ⇐ a(hotel, Hotel) then
            booking(Total-Cost) ⇒ a(client, Client)
                ← is(Total-Cost, Flight-Cost + Hotel-Cost)   ) or
          ( no-vacancy ⇐ a(hotel, Hotel) then
            failure ⇒ a(client, Client)                      )
        )
a(hotel, Hotel) ::
        book-hotel(Location, Start, End, Money) ⇐ a(Role, Agent) then
        room-available(Location, Start, End, Money, Cost) ⇒ a(Role, Agent)
            ← room-available(Location, Start, End, Money, Cost) or
        no-vacancy ⇒ a(Role, Agent)

a(airline, Airline) ::
        book-flight(Src, Dst, Start, End, Money) ⇐ a(Role, Agent) then
        flight-available(Src, Dst, Start, End, Money) ⇒ a(Role, Agent)
            ← flight-available(Src, Dst, Start, End, Money) or
        no-flights ⇒ a(Role, Agent)

a(matchmaker, matchmaker) ::
        record-matchmaking-outcome(Outcome)
            ← matchmaking(Outcome) ⇐ a(Role, Agent)
```

service providers. Further, we must ask how many service types will be provided. Again, in each domain, we might have a simple, monolithic interface, or an interface with such fine granularity that few engineers ever fully understand or exploit it. The answers to these will impact the nature of our matchmaking infrastructure.

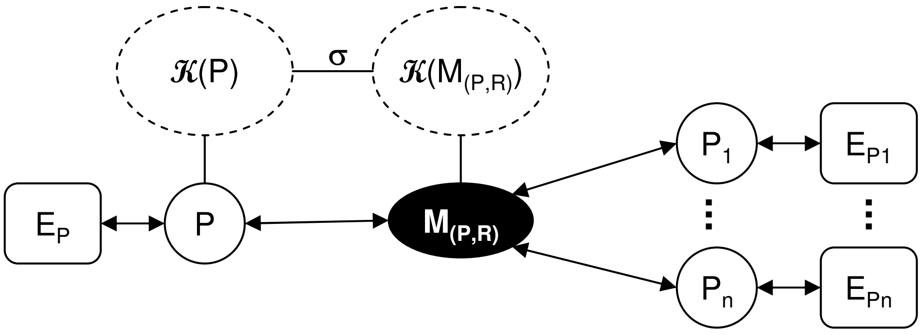
While our technique handles large numbers of incidences, it does not scale for very large numbers of services or roles. For any protocol with a set of roles R, and with each role having $|providers(r_i)|$ providers, the number of ways of choosing a team is $\prod_{r_i \in R} |providers(r_i)|$, or $O(m^n)$. No matchmaking system could possibly hope to discover all the various permutations of services in a rich environment, although machine learning techniques might be helpful in directing the search for groupings of services. How much of an issue this actually becomes in any particular domain will be heavily influenced by the outcomes to the issues discussed above.

3 Knowledge Consistency Between Peer and Interaction Model

A peer in an open knowledge sharing system may need to participate in an interaction that it has not experienced hitherto, or it may need to resume an interaction in which it had participated some time ago. The deployment mechanisms used for knowledge sharing in the OpenKnowledge project allow a formal specification of the current state of a given interaction (new or continuing) to be communicated to a peer so, in the terminology used in Definition 1, the peer has an expicit representation of $M_{(P,R)}$. It does not necessarily know, however, whether its participation in the interaction coordinated via $M_{(P,R)}$ will maintain consistency between its local knowledge and that of the interaction model. The property we would ideally preserve is described as Property 2 below.

Property 2 *If proposition X_1 is in the knowledge set of P and proposition X_2 is in the knowledge set of interaction model $M_{(P,R)}$ at any time then X_1 and X_2 should be consistent.*

$$X_1 \in \mathcal{K}(P) \ \wedge \ (X_2 \in \mathcal{K}(M_{(P,R)}) \ \vee \ \diamond(X_2 \in \mathcal{K}(M_{(P,R)}))) \quad \rightarrow \quad \sigma(X_1, X_2)$$



Property 2 cannot be guaranteed in general because, in general, we are not guaranteed to be able to enumerate all the elements of $\mathcal{K}(P)$ or $\mathcal{K}(M_{(P,R)})$. Furthermore, we require consistency not only for the current state of the interaction but for all its future states. This guarantee into the future cannot be made on a specific peer because the future interaction state may be determined by other peers of which it has no knowledge.

Given that we cannot solve the general problem, one established method of dealing with it is by specifying the main local constraints the acceptability of a particular role by a peer (a subset of $\mathcal{K}(P)$ and proving that these are not violated by anything that currently can be temporally inferred from the interaction model. Although much of a peer's knowledge is private, there are situations in practice when peers make explicit a portion of their knowledge in the form of constraints on the interactions they will allow with other peers. For example, a peer might describe permissions (constraints on which peers it will interact with and when); obligations (constraints assumed on peers with which it interacts); or trust policies (constraints on the degree to which it is prepared to rely on other peers). Various formal languages have been developed for expressing this sort of knowledge (for instance ASL [5], RDL [4], and Rei [6]). In [7] we show in detail how to combine this sort of deontic specification with LCC and demonstrate how this can establish a measure of trust in interaction models. For compactness, we do not repeat this result here but give instead the framework used, which is based on a from of model checking.

3.1 The Verification Process

The model checker is built on top of XSB tabled prolog. This means that calls to given predicates will be cached, along with their answers, in a *table*. This collection of tabled calls paired with their answers is consulted every time a new call is issued. Here is how this works. When a call is made, the following cases are tested:

1. *If the new call matches a tabled one:*
   The set of answers associated with this tabled call are retrieved and the new call is resolved against these answers.
2. *f the new call does not find a match against the tabled calls:*
   The new call is entered into the table and is resolved against the Prolog program clauses. For each answer derived during this process, the following cases are tested:
   (a) *if the answer is not already in the table:*
   The new answer is inserted into the table entry associated with this call.
   (b) *if the answer already exists in the table:*
   The evaluation simply fails and backtracks to generate more answers.

Notice that the calls are resolved against unique answers instead of repeated program clauses. This process terminates when the call matches a finites number of defined Prolog program clauses and each of these result in a finite number of answers.

In our case, the main question the model checker tries to answer is whether a given temporal property is satisfied in a given (interaction's) state-space. The model checker starts by verifying that the temporal property is satisfied at the initial state of the state-space. For this, the `satisfies` predicate, which specifies the μ-calculus proof rules (Figure 8) in Prolog, is called. The predicate is a tabled predicate. This means that every time a call is made to this predicate when a state needs to be verified against a given temporal property, the first thing to do is to search the table for answers. Only if this is a new call — a call with a new combination of a state and a temporal property — it will be resolved against the `satisfies` Prolog predicates. In practice, this implies that the model checker will always terminate with an answer in a finite state-graph[1].

[1]Note that while the state-graph is finite, the state-space may be infinite. This happens when one state in the state-graph has a transition that connects it to a previous state, resulting in an infinite state-space to be generated from infinite copies of finite states.

$$\begin{array}{lll}
\texttt{satisfies}(E,\texttt{tt}) & \leftarrow & \texttt{true} \\
\texttt{satisfies}(E,\phi_1 \vee \phi_2) & \leftarrow & \texttt{satisfies}(E,\phi_1) \vee \texttt{satisfies}(E,\phi_2) \\
\texttt{satisfies}(E,\phi_1 \wedge \phi_2) & \leftarrow & \texttt{satisfies}(E,\phi_1) \wedge \texttt{satisfies}(E,\phi_2) \\
\texttt{satisfies}(E,\langle A\rangle\phi) & \leftarrow & \exists F.\ ((E \xrightarrow{A} F)\ \wedge\ \texttt{satisfies}(F,\phi)) \\
\texttt{satisfies}(E,[A]\phi) & \leftarrow & \forall F.\ ((E \xrightarrow{A} F)\ \rightarrow\ \texttt{satisfies}(F,\phi)) \\
\texttt{satisfies}(E,\mu Z.\phi) & \leftarrow & \texttt{satisfies}(E,\phi) \\
\texttt{satisfies}(E,\nu Z.\phi) & \leftarrow & \mathit{dual}(\phi,\phi') \wedge \neg\texttt{satisfies}(E,\phi')
\end{array}$$

The rules imply that a state E always satisfies `tt` (*true*), and never `ff` (*false*). E satisfies $\phi_1 \vee \phi_2$ if it satisfies either ϕ_1 or ϕ_2, and it satisfies $\phi_1 \wedge \phi_2$ if it satisfies both ϕ_1 and ϕ_2. $[A]\phi$ is satisfied if for all transitions A that state E can take to F, then F satisfies ϕ. $\langle A\rangle\phi$ is satisfied if state E can make at least one transition A to state F, such that F satisfies ϕ. Prolog, by nature, computes the least fixed point solution. Hence, $\mu Z.\phi$ is satisfied if state E satisfies the property ϕ. The greatest fixed point, however, is the dual of the least fixed point. Therefore, the greatest fixed point formula is satisfied if the least fixed point of the negated formula fails to be satisfied.

Figure 8: The μ-calculus proof rules

If the predicate is not satisfied at the initial state, a transition is then made to the next state(s) in the state-space. The transition is computed by calling the `transition` predicate, which specifies the LCC transition rules (Figure 9). The property is then verified against the new state(s), as the μ-calculus proof rules indicate. The model checker should also make sure that each transition made does not break the agents' specified deontic constraints.

3.2 The Model Checking Framework

The framework of our model checker is presented in Figure 10. The model checker itself is built on two modules based on the temporal language's proof rules as well as the process calculus' transition rules, which the proof rules require. Note that in our implementation, the temporal language used is the μ-calculus and the process calculus in the lightweight coordination calculus (LCC).

The model checker is built on top of the XSB tabled Prolog system. The XSB Prolog engine is called to verify that a certain temporal property is satisfied in a given state-space. Verification is carried on based on the μ-calculus proof rules, specified in Prolog via the `satisfies` predicate. This requires knowledge of the temporal property to be verified and the interaction's state-space it is verified upon. During the verification process, a table containing cached results of previous calls is consulted. As mentioned in the section above, verification might require a transition to be made from one state in the state-space to another. In such a case, the XSB Prolog engine computes the transition based on the LCC transition rules, specified in Prolog via the `transition` predicate. This requires knowledge of the state-space as well as the deontic constraints, to make sure transitions do not break these constraints.

4 Knowledge Consistency Between Interacting Peers

In Section 3 we discussed the issue of consistency between peer ans interaction model. We now turn to the issue of consistency between peers. This is a difficult practical problem, even with our explicit interaction models, because the vast majority of knowledge local to a peer ($\mathcal{K}(P_i)$) at any time is private to that peer.

$$\frac{}{M \Leftarrow A \xrightarrow{in(M)} nil} \qquad \frac{}{(A \leftarrow C) \xrightarrow{\#(X)} A}\ sat(C) \wedge X\ in\ C$$

$$\frac{}{M \Rightarrow A \xrightarrow{\overline{out}(M)} nil} \qquad \frac{A \xrightarrow{a} E}{(A \leftarrow C) \xrightarrow{a} E}\ sat(C) \wedge (a \neq \#/\_)$$

$$\frac{}{null \xrightarrow{\#} nil} \qquad \frac{A \xrightarrow{a} E}{A\ par\ B \xrightarrow{a} E\ par\ B}$$

$$\frac{B \xrightarrow{a} E}{A \xrightarrow{a} E}\ A ::= B \qquad \frac{B \xrightarrow{a} E}{A\ par\ B \xrightarrow{a} A\ par\ E}$$

$$\frac{A \xrightarrow{a} E}{A\ or\ B \xrightarrow{a} E} \qquad \frac{A \xrightarrow{a} E \qquad B \xrightarrow{\overline{a}} F}{A\ par\ B \xrightarrow{\tau} E\ par\ F}$$

$$\frac{B \xrightarrow{a} E}{A\ or\ B \xrightarrow{a} E} \qquad \frac{A \xrightarrow{a} nil}{A\ then\ B \xrightarrow{a} B}$$

$$\frac{A \xrightarrow{a} E}{A\ then\ B \xrightarrow{a} E\ then\ B}\ E \neq nil$$

For the agent to perform a transition step, the transition rules above 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 ($\#$ represents 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.
Note that all $in(M)$, $\overline{out}(M)$, and $\#$ transitions should be checked to make sure they do not break any deontic rules (i.e. the transitions satisfy the deontic constraints).

Figure 9: The LCC transition rules

We can make use of whatever guarantees are possible pairwise between peers and interaction models (see Section 3) to attack part of this problem but that addresses only knowledge that is explicitly shared during the interaction. Fortunately, for many (perhaps the majority) interactions the issue of consistency is not as broad as that stated in Property 3 below, since we require consistency only as far as local knowledge related to the interaction is concerned.

Property 3 *If proposition X_1 is in the knowledge set of P and proposition X_2 is the knowledge set of process P_i then X_1 and X_2 should be consistent.*

$$X_1 \in \mathcal{K}(P) \ \wedge \ P_i \in \mathcal{P}(M_{(P,R)}) \ \wedge \ X_2 \in \mathcal{K}(P_i) \quad \rightarrow \quad \sigma(X_1, X_2)$$

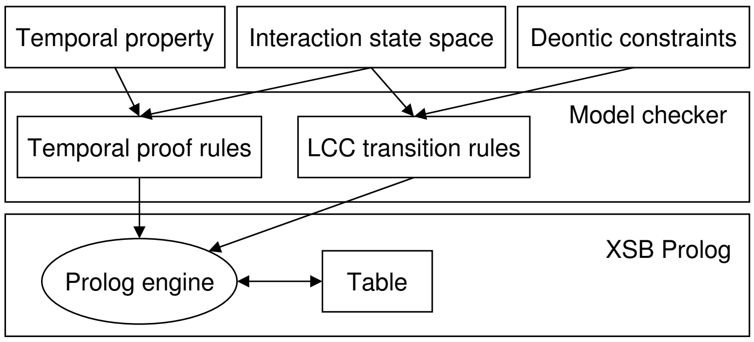
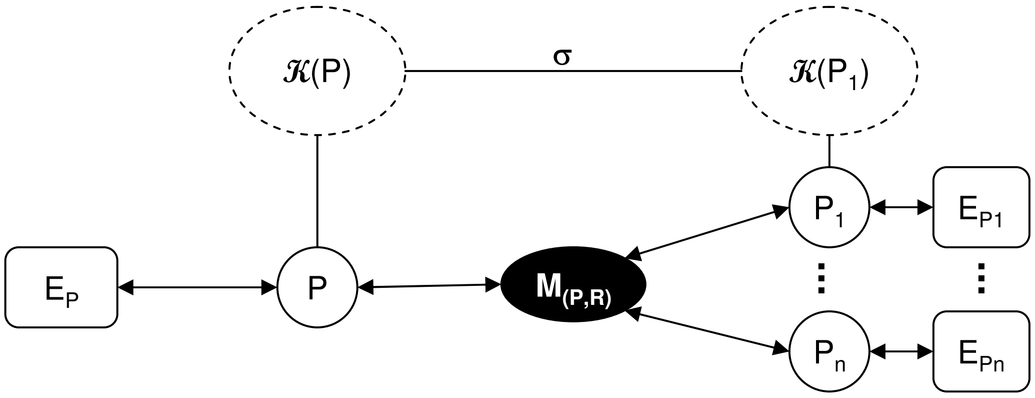


Figure 10: The system's architecture



One of the things that makes consistency difficult to achieve between peers is that the content of messages exchanged during interactions is often bound by implicit constraints. In particular, the context of the interaction influences heavily the content of the messages. There are different levels of context:

- the specific topic of the interaction: for example if the interaction is about the purchase of a camera, then most of the messages will be about this specific subject
- the geographical/economical/historical context: a talk about a digital camera in 1990 was rather unusual, while it is more common now. A talk about the weather is more common in the UK rather than in south Spain, and it will reflect different issues (wind, cold, rain or sun and drought)

Moreover, interactions follow paths constrained by the interaction model ($M_{(P,R)}$). For instance if the buyer has accepted an offer, it is then supposed to pay (by, say, giving credit card details), or if the buyer has asked about the price, the vendor is supposed to say a price.

The work presented in [8] aims at extracting, and then using the information embedded in the interactions between agents. The knowledge of this information can be exploited to predict the content of a message, given the current state of the interaction. The prediction of the message content can be used to select the most likely mappings for terms defined in other ontologies. The agent will still need to apply some ontology mapping technique to select the best mapping term from the suggestions, but the search space is greatly reduced by the use of the information extracted from the unfolded dialogue. The suggestions can also help to reduce the ambiguities that a mapping process, unaware of the context of use of terms, may not be able to solve.

In the remainder of this section we present an evaluation framework for the system discussed above. Section 4.1 presents the work to be evaluated. Section 4.2 describes how the evaluation is performed, starting from an overview of the implementation chosen for the system, and then detailing the methodology used for testing.

4.1 Learning and predicting the possible message contents

4.1.1 Background assumptions

As described in [10], agent interactions can follow a *mentalistic* approach - where every agent needs to model the other agents intentions and beliefs in order to plan the conversations - or a *social* approach, where the focus is on the rules and conventions that the agents need to follow. In the social approach, the agents usually follow some sort of interaction model, that describe the moves given the current state. The moves are usually the messages that can be sent or that can be expected, often constrained by some specific rules.

A message is usually a tuple, whose elements convey the content of the single communication act:

$m_i = \langle e_1, ..., e_n \rangle$

The element e_i in the message refers to some conceptual entity, represented with some symbol s_i belonging to the agent that introduces the term. If all agents share the same ontology, all the symbols are understood by all the agents. If this is not the case, then the agents have to use an "oracle" to map the symbols to the correct entities.

Let's suppose that an agent, with ontology L_a, receives a message $m_k(\ldots, w_i, \ldots)$ when in a specific state of an interaction, and that $w_i \notin L_a$ is the foreign term. The task of finding what entity or concept, represented in the agent's ontology by the term t_m, was encoded in w_i by the transmitter is performed by some "oracle", whose actual implementation and method is not relevant for the work. Not all the comparisons between w_i and terms $t_j \in L_a$ are useful: the aim of the work evaluated here is to specify a method for choosing the smallest set $\Lambda \subseteq L_a$ of terms to compare with w_i, given a probability of finding the matching term $t_m \in L_a$. We assume that t_m exists and that there is a single best match.

Let $p(t_j)$ be the probability that the *entity* represented by $t_j \in L_a$ was used in w_i inside m_k. The oracle will find t_m if $t_m \in \Lambda$, event that has a probability:

$$p(t_m \in \Lambda) = \sum_{t_j \in \Gamma} p(t_j) \tag{3}$$

As shown in figure 11, if all terms are equiprobable, then $p(t_m \in \Lambda)$ will be proportional to $|\Lambda|$. For example, if $|L_a| = 1000$, then $p(t_j) = 0.001$. Setting $|\Lambda| = 800$ yields $p(t_m \in \Gamma) = 0.8$, and there is no strategy for choosing the elements to add to Λ.

Instead, if the probability is distributed unevenly, and we keep the most likely terms discarding the others, we can obtain a higher probability for smaller Λ. For example, suppose that $p(t_j)$ is distributed approximately according to Zipf's law (an empirical law mainly used in language processing that states that the frequency of a word in corpora is inversely proportional to its rank):

$p(k; s; N) = \frac{1/k^s}{\sum_{n=1}^{N} 1/n^s}$

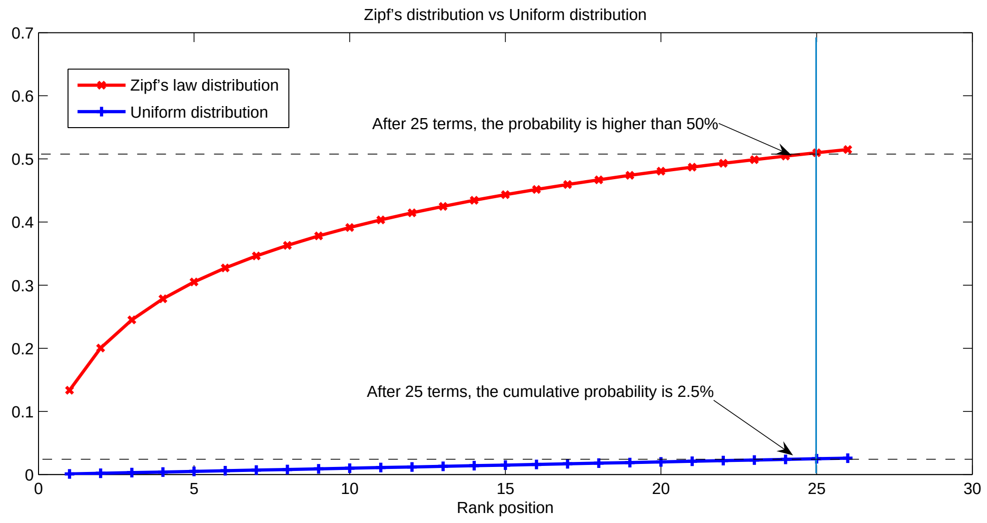


Figure 11: Zipf's law distribution vs uniform distribution

where k is the rank of the term, s is a parameter (which we set to 1 to simplify the example), and N is the number of terms in the vocabulary. For $|L_a| = 1000$, then $p\,(t_m \in \Lambda) = 0.70$ for $|\Lambda| = 110$ and maybe more remarkably $p\,(t_m \in \Lambda) = 0.5$ for $|\Lambda| = 25$.

Therefore, given a probability distribution for the terms, it is possible to trade off a decrement in the probability of finding the matching term t_m in Λ with an important reduction of comparisons made by the oracle.

The issue dealt by this paper is to verify that the theoretical probability $p\,(t_m \in \Lambda)$ is reached consistently by the model described in [8], that means that repeating the interaction over and over, the correct entity for a slot W_i in a message is contained in Λ with the frequency specified by the theoretical probability.

4.1.2 Basic model description

In [8], the suggested solution is a model of the interaction in which the frequencies of the properties of the entities used to instantiate each variable W_i in different runs of the same protocol are stored and updated.

In general, the possible values for a *slot* W_i in a message are modelled by M assertions, each keeping track of the frequency with which the matching entity for the slot has been part of a set Ψ in the encountered dialogues:

$$A_j \doteq Freq\,(slot\_value \in \Psi) \tag{4}$$

Assertions can simply be about the frequency of entities in a slot, disregarding the values of other slots in the protocol run:

$A_j^{\langle \mathtt{N_i,a}\rangle_{\mathtt{R}}} \doteq \mathit{freq}\,(\mathtt{slot\_value} \in \{e_q\}) = p_j$
More precise assertions can be about the frequency of an entity given the values of previous slots:
$A_j^{\langle \mathtt{N_i,a}\rangle_{\mathtt{R}}} \doteq \mathit{freq}\,(\mathit{slot\_value} \in \{e_q\} \mid \mathit{slot}_{\mathtt{R}} = e_k) = p_j$
Assertions can also be about ontological relations between the entities in the slot and other entities. The possible relations depend on the expressivity of the ontology: if it is a simple list of allowed terms, it will not possible to verify any relation; if it is a taxonomy, subsumption can be found; for a richer ontology, more complex relations such as domain or range can be found. Assertions about ontological relations are obtained generating hypotheses about different relations and keeping the count of the proved ones.

The hypotheses can be about an ontological relation between the entity in the slot and an entity e_k in the agent's ontology:
$A_j^{\langle \mathtt{N_i,a}\rangle_{\mathtt{R}}} \doteq \mathit{Pr}\,(\mathit{slot\_value} \in \{X | \mathit{rel}\,(X, e_k)\}) = p_j$
The assertions can also regard the relation with another slot in the protocol:
$A_j^{\langle \mathtt{N_i,a}\rangle_{\mathtt{R}}} \doteq \mathit{Pr}\,(\mathtt{slot\_value} \in \{X \mid \mathit{rel}\,(X, \mathtt{previous\_slot\_value})\}) = p_j$

The assertions can be generated using different strategies, and assign probabilities to overlapping sets that can be either singletons or larger. The aim of this system is to select the most likely entities for a slot in order to reach a given probability of finding the mapping, and therefore we need to assign to the terms the probabilities computed with the assertions.

This requires two steps. First, probabilities given to sets are uniformly distributed among the members: according to the *principle of indifference*, the probability of mutually exclusive elements in a set should be evenly distributed. Then, the probability of an entity t_i is computed by summing all its probabilities, and dividing it by the sum of all the probabilities about the slot:

$$p\,(t_i) = \frac{\sum A_j^{\langle N,A\rangle_R}\big(\langle N,A\rangle_R \in \{t_i\}\big)}{\sum A_k^{\langle N,A\rangle_R}} \tag{5}$$

The probabilities of all the terms are sorted by probability, and Λ will be created adding terms until the cumulative probability exceeds the specified threshold.

4.2 Testing

In order to test and evaluate the feasibility and the reliability of the model, I developed a framework that can run different dialogues, analysing the message content in order to create models for the interactions, and then applying them to predict the content of messages in similar interactions.

4.2.1 Interaction Framework

As the focus of the problem is on agents' interactions, and particularly conventional interactions, the the framework is built around a model of interaction derived from the concepts of social rules. The agents involved in dialogues use the Lightweight Coordination Calculus (LCC).

The *Lightweight Coordination Calculus* (LCC) is an executable specification language adapted to peer-to-peer workflow and has been used in applications such as business process enactment [3] and e-science service integration [1].

LCC is based on process calculus: protocols are declarative scripts written in Prolog and circulated with messages. Agents execute the protocols they receive by applying *rewrite rules* to expand the state and find the next move.

Role r_b clause

$\texttt{a(}r_a(O)\texttt{, I) ::=}$
$m_1(X) \Rightarrow \texttt{a(}r_b\texttt{,O)} \leftarrow \kappa_1(X)$
$\texttt{then}$
$m_2(Y) \Leftarrow \texttt{a(}r_b\texttt{,O).}$

Role r_b clause

$\texttt{a(}r_b\texttt{, O) ::=}$
$m_1(X) \Leftarrow \texttt{a(}r_a(\_)\texttt{, I)}$
$\texttt{then}$
$m_2(Y) \Rightarrow \texttt{a(}r_a(\_)\texttt{, I)} \leftarrow \kappa_2(X, Y)\texttt{.}$

Figure 12: Simple LCC protocol

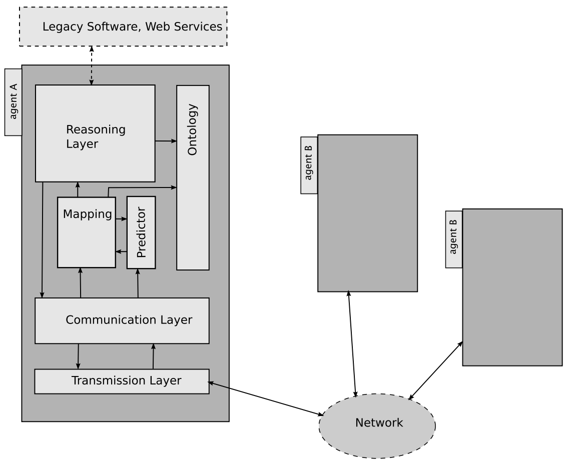


Figure 13: Agent architecture

It uses roles for agents and constraints on message sending to enforce the social norms. A role behaviour is defined by a clause, and the basic behaviours are to send ($\Rightarrow$) or to receive ($\Leftarrow$) a message. More complex behaviour are expressed using connectives: `then` creates sequences, `or` creates choices. Common knowledge can be stored in the protocol.

Figure 12 shows a simple protocol, where the message $m_1(X)$ is sent to the agent that plays role r_b, after satisfying the constraint $\kappa_1(X)$, and then the agent impersonating role r_b replies with the message $m_2(Y)$, after satisfying the constraint $\kappa_2(X, Y)$.

Agents in the framework are composed by layers, as shown in figure 13. The lower level is the *transmission layer*, that manages the transmission and reception of the messages over the network. The *communication layer* wraps the LCC engine and processes the received messages, together with the protocol sent along with them. As we have seen before, protocols require constraints to be satisfied as preconditions to message sending or postconditions to message reception: the constraints, such as $\kappa_1(X)$ and $\kappa_2(X, Y)$ in the example, are solved calling the *reasoning layer* that wraps all the agent specific skills and knowledge. Interactions take place in an open environment, where agent may not share the same ontology. Therefore the reasoning and the communication layers are connected through a *mapping layer*. Terms received in the messages can be used in constraints (for example, in the protocol in figure 12, in the constraint $\kappa_2(X, Y)$ the content of X was defined by another agent), and the mapping layer tries to find the equivalent (or the closest) terms in the agent's ontology, and translate the constraints transparently for the reasoning layer. The agent's *ontology layer* provides to these two layers the methods to reason over the agent ontology.

The *predictor layer* analyses the content of the received messages, of the satisfied constraints, and of the role calls, creating a model of the interaction that uses in suggesting the possible content of the received messages in similar interactions.

4.2.2 Approach to Testing

The aim of testing, as stated before, is to verify how often the predictor suggests a set of terms that contains the correct mapping, and how large are these sets, with the theoretical ideal being as described in section 4.1.1: if the correct terms appear in the set Λ less often than the probability sum in expression 3 predicts, then the probability distribution created by the model is imprecise, or the distribution of the modelled phenomenon varies with time. If the suggestion sets are too large, then the computed probability distribution is too uniform, and either it was not possible to extract any meaningful relation between terms in the dialogue, or there are none.

The testing could be made through real interaction scenarios, using real ontologies and real workflows for the dialogues. However, this would cover only part of the testing space, without having the possibility of varying parameters to verify the effects.

A more effective approach for testing is having a set of protocols that cover different types of abstract relations between the messages, reflecting different real cases, a set of ontologies generated according to some parameters, and algorithms for generating choices according different probability distributions. An argument supporting this approach is that agents do not understand the meaning of the conversations: they can only count frequencies and figure out relations between terms in the messages, and therefore having words (meaningful for us, but just strings for the agents) does not really change the testing.

For example, a simple protocol as the one described in the example (figure 12) can be about an agent telling x to a second agent, that replies with y, related to x. The term x is chosen from the first agent's ontology according to some particular preference, as we will see in the next section. This is a rather realistic abstraction: vendor receives more requests about some products than about others. The term y is ontologically related to the term x' equivalent to x in the second agent's ontology. It can be a superclass, a subclass, an instance, a sibling or a property.

The agents, running this simple protocol many times should learn the probability distributions of the terms in x and y and predict, with increasing precision the set of terms that could appear in them. Obviously, the second agent has the most difficult task, being able to predict only the prior probability of x, while the first agent should do better, knowing x, in predicting what could be y.

4.2.3 Protocols for Testing

We have created a number of protocols, that reflect different patterns in dialogues and in relations between the content of messages. Most of the protocol are based on the simple *tell*/*reply* model seen for the dialogue in figure 12. More complex protocols include the possibility of constraint failure, and therefore the possibility that different messages are sent (and received) in a particular state of the interaction. Other complex protocols exploit the recursion available in LCC.

The constraints in the protocol, like κ_1 and κ_2 in the example, simulate real world constraints. Some of these constraints, like κ_1 in the example, simulate constraints used to express a preference in real world protocol, like what product to buy from a vendor. In the real world, preferences reflect the preferences of a number of users, and normally have skewed distributions: some elements are more requested than others. During the testing process, the interaction is repeated over and over between two agents: agents are given probability distributions for the constraints. At every run of the protocol the constraint will be satisfied with a different element - according to its specific distribution.

A preference distribution δ is a list L of terms taken from the agent's ontology, in some arbitrary order together with a probability distribution D used to generate a number between 0 and the size of the list. The number is used as index to extract a term from the list. For example, the probability distribution can be the Gaussian one. In this case, the variance parameter specifies how spread the distribution must be: a very narrow curve means that only a few terms will be chosen, making the content of the message easily predictable, while a wider curve means that many terms can be chosen, increasing the uncertainty about the possible content of the message.

Other constraints, like κ_2 in the example, simulate real world constraints that search elements related to the ones given as input: by convention, when asked a question an agent replies with something related to the question, possibly according to its preferences. The constraint first finds all the elements that satisfy the specific relation for the protocol, and then an element can be chosen using some probability distribution to simulate a preference.

4.2.4 Ontologies

The terms used in each protocol are obtained either by preference distribution over an ontology, or by searching related terms in the ontology. The ontologies are generated as graphs, composed by a main tree, that correspond to the class taxonomy plus the instances, and links between the class and instances nodes that represent the properties. The taxonomy tree can be generated with different depths and different average number of children per node. The max (or average) number of properties can be set. Playing with these parameters is possible to emulate flat lists, without hierarchy, flat ontologies, with light hierarchy, or more hierarchical structures. See figure 14 for an example of generated taxonomy.

The goal of the experiments is to verify how well the agents can predict the content of the messages: therefore there is no specific need to have different ontologies: experiments can be run with the agents sharing the same ontologies. However, it is possible to apply a set of transformations to an ontology, obtaining automatically two different ontologies and the mappings between the two. The mappings are used instead of the mapping oracle, as I am not interested in verifying the quality of a mapping process.

Using different ontologies means that some of the relations between the terms in the messages will not exist for both agents, making the prediction more difficult and less reliable.

4.2.5 Experiment runner

The experiments consist in running repeatedly the protocols (as described above) the constraints of which are satisfied using probability distributions to simulate a large population of agents.

The experiments are run through a framework that parses XML files describing the experiments, instantiates the component needed to perform the interactions, and starts the dialogues as many times as requested. The experiment batch XML file describes a set of experiment to run. It first lists the agents involved in the different experiments, and then, for each experiment, it defines the values for some agents parameters (so that the same agents can show different behaviours in different experiments), specifies if the internal state of

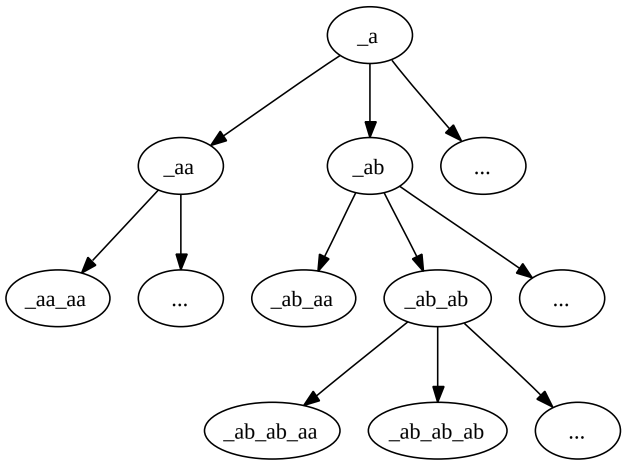


Figure 14: A generated ontology

the agents must be reset before starting the experiment, and defines what protocol must be run and for how many time, defining also the protocol parameters. Through the agent parameters it is possible to specify which prediction strategies should be used (see section 4.1.2), allowing a comparison between them.

It is also possible to create new experiments without the need to specify all the parameters: an experiment can derive from another one. Only the different parameters need to be specified.

The file in figure 15 describe two experiments using the protocol in figure 12. The only difference between the two experiment, both involving 50 repetitions of the interaction, is in the variance of the Gaussian distribution: the first curve is narrower than the second.

The results of running this batch of experiments is shown in figures 16 and 17. The tables shows, for each of the two agents involved, the features of the suggested set Λ after 20, 40 and 50 iterations of the protocol. The score represent how often the suggested set Λ contained the correct term.

The learnt model, for the agent interpreting role r_a in the dialogue and using the ontology in figure 14, is shown in figure 1. As seen before, an assertion is generated and maintained by a specific strategy and states how many times the content of the slot in a message was in the set defined in the assertion.

5 Ensuring Knowledge Consistency Between Peer and Environment

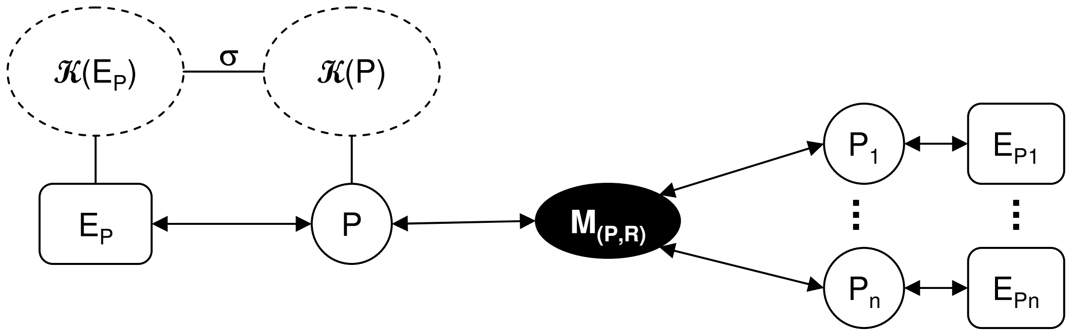
Perhaps the most difficult practical issue of all, from an analytical point of view, in distributed knowledge sharing is that of ensuring consistency between the knowledge available to an interacting peer and the knowledge available in its environment. This is because a peer's environment is likely to include some part of the physical world, thus analytical methods that apply here must claim relevance to that world in all its complexity.

Property 4 *If proposition X_1 is in the knowledge set of peer P and proposition X_2 is the knowledge set of its environment, E_P, then X_1 and X_2 should be consistent.*

```
<batch>
 <description>use of protocol 1</description>
 <involved_agent id="tagent1"/>
 <involved_agent id="tagent2"/>
 <experiment id="1">
   <description>Learn the distribution of a variable (with sigma=15)</description>
   <reset agent="tagent1"/>
   <reset agent="tagent2"/>
   <agent_param agent="tagent1" section="general" param="feedback_results" value="true"/>
   <agent_param agent="tagent1" section="randprefs" param="totell" value="{'file':'t1pa', 'sigma':15}"/>
   <institution name="prot1" repeat="50" dumpevery="20">
    <start role="r1a" agent="tagent1">
     <param>tagent2</param>
    </start>
   </institution>
 </experiment>
 <experiment id="2" derived_from="1">
  <description>Learn the distribution of a variable (with sigma=5)</description>
  <reset agent="tagent1"/>
  <reset agent="tagent2"/>
  <agent_param agent="tagent1" section="randprefs" param="totell" value="{'sigma':5}"/>
 </experiment>
</batch>
```

Figure 15: XML file describing an experiment

$$X_1 \in \mathcal{K}(P) \ \wedge \ X_2 \in \mathcal{K}(E_P) \quad \rightarrow \quad \sigma(X_1, X_2)$$



Although there is no possibility of verifying that, in general, knowledge is consistent across environments for interacting peers, we can investigate whether specific forms of interaction maintain specific forms of consistency for specific types of environment. By narrowing our focus in this way we can take advantage of specific forms of environmental simulation. As an example of this we have chosen Unreal Tournament.

Unreal Tournament is a multi-agent gaming environment where multiple players compete in a 3D world in order to achieve certain goals. These goals are defined by the type of game being played, where game-

`Results for agent impersonating` r_a

	20	40	50
average size	13.2	14.5	14.8
std dev	4	3.1	2.83
max, min size	[17,0]	[17,0]	[17,0]
score	.55	.68	.68

`Results for agent impersonating` r_b

	20	40	50
average size	33	44.6	48.5
std dev	15.4	15.9	16.3
max, min size	[53,0]	[60,0]	[65,0]
score	.25	.38	.48

Figure 16: Results of running the experiment with $\sigma = 15$

`Results for agent impersonating` r_a

	20	40	50
average size	12.8	14.5	15
std dev	3.3	2.89	2.77
max, min size	[15,0]	[17,0]	[17,0]
score	.60	.62	.62

`Results for agent impersonating` r_b

	20	40	50
average size	32	46.4	50.7
std dev	13.8	17.8	18.1
max, min size	[49,0]	[68,0]	[69,0]
score	.25	.35	.44

Figure 17: Results of running the experiment with $\sigma = 5$

types include co-operative team games, such as Capture the Flag. Typically games in Unreal Tournament are combative but this need not be the case, and the most interesting aspect of the environment for our purposes is that it allows complex simulated physical environments and complex autonomous players to be constructed. It also provides a sophisticated graphics engine that allows us to form opinions on the human realism of the simulations being run. To achieve this sort of sophistication it is a complex software package but (via the open source GameBots protocol) it permits any language with TCP/IP capabilities to send messages to the Unreal Tournament game to control the actions of a player. Using this, we have built a mechanism for communicating between Unreal Tournament games and an LCC interpreter, so Unreal Tournament supplys the (simulated) source of $\mathcal{K}(E_P)$ in analysing Property 4.

The system was built around the idea that all the bots should be multiple running Java threads with identical baseline functional abilities within the Java itself. In order to use these functional abilities each bot would have access to a particular LCC protocol. At every time-step (each time a message was received from GameBots , roughly 4-5 times a second) the bot would then pass all its personal game information to a LCC interpreter (written in Prolog) along with its LCC strategy. The return from this would be a decision about what the bot would then do. Changes in the bot's behaviour could then be attributable to a combination of

Assertion	Origin
`P(set(["'_ab_ai'"])|None)=3`	`termfilter`
`P(set(["'_ab_aj'"])|None)=4`	`termfilter`
`P(set(['root'])|None)=1`	`termfilter`
`P(('getSuperclasses', <r2a,satisfied,totell_1,0>)|None)=50`	`relationfilter`
`P(set(["'_ab'"])|None)=9`	`termfilter`
`P(('subClassOf', '_ab')|None)=40`	`ontoanalysis`
`P(set(["'_ab_ae'"])|None)=1`	`termfilter`
`P(set(["'_ab_ag'"])|None)=2`	`termfilter`
`P(set(["'_ab_ad'"])|None)=11`	`termfilter`
`P(set(["'_ab_ab'"])|None)=7`	`termfilter`
`P(set(["'_ab_ah'"])|None)=7`	`termfilter`
`P(('subClassOf', 'root')|None)=9`	`ontoanalysis`
`P(set(["'_ab_af'"])|None)=3`	`termfilter`
`P(set(["'_ab_ac'"])|None)=2`	`termfilter`

Table 1: Model learnt by `tagent1` for `reply(X)`

changes in the game-state and the rules in its LCC strategy.

As an example of this method in use we summarise below an analysis performed with the simulator when running a variation of the popular Team Death Match game-type (in which one team of gamebots attempts to kill as many of the other team as possible). This was used primarily as a testing scenario for calibrating the bots' firing capacities to that of the in-built bots but revealed some interesting results when the LCC strategies were also ran with it. It also allowed for more empirical evidence to be gathered for some strategies as the games were much shorter and multiple trials could be run. In all the trials in this section the bots were played against in-built enemy bots set on very high levels of skill (in terms of their shooting , pathing and various other autonomous abilities). The enemy bots were also set shoot at anything on an opposing team which strayed into their vision. The game stopped when one team got to 60 kills, this team were thus the winning team. Several trials were run using different strategies. Each trial was run 10 times. All trials in this section were ran on the same (Gael) environment map.

The results are summarised in the chart of Figures 5 and 5. Eight different trials are summarised in terms of the number of kills made by the LCC coordinated team and the enemy team Figure 5 and the number of wins obtained by the LCC coordinated team and the enemy team Figure 5. Note that the average number of wins is not directly proportional to the average number of kills in any trial. The conditions for each trial are given below.

Trial 1 : The first experiment was a baseline setup. This was done by playing one in-built bot against one bot running the following LCC strategy, which requires the bot simply to run around randomly, making no attempt to follow or approach its enemy:

$$a(random, R) :: null \leftarrow movementAttempt(random\_play)$$

Trial 2 : This trial is similar to Trial 1 except that three LCC bots running the strategy from Trial 1 were played against the single enemy bot. This was to try to determine the effect that more team members

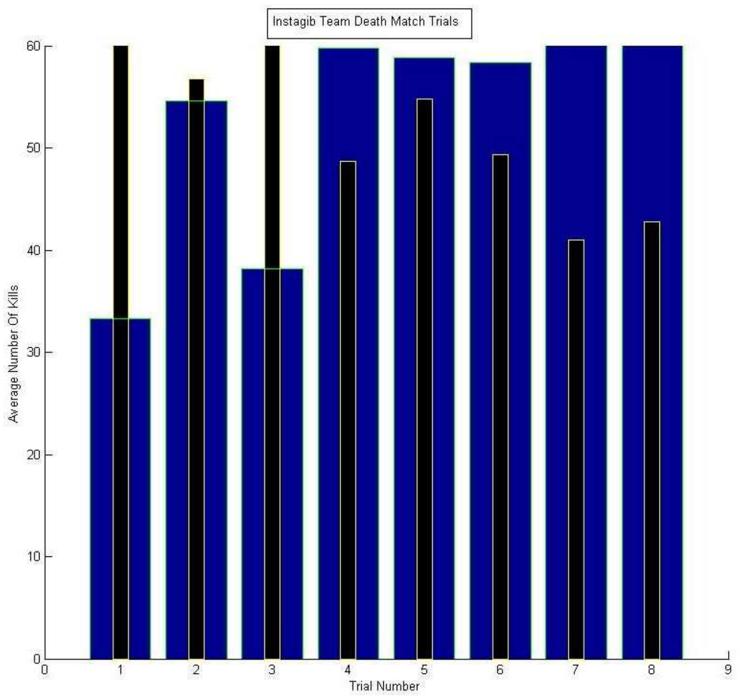


Figure 18: Number of kills for each trial (LCC bots in blue, enemy bots in black)

had on performance. Having more team members does not drastically help performance of the bots without more sophisticated coordination.

Trial 3 : In this trial one bot running the following strategy was played against one in-built bot, where $visiblePlayer(L)$ is true if a player is visible to the bot at location L and $movementAttempt(L)$ is true if the bot attempts to move to location L.

$$
\begin{aligned}
&a(follower, R) :: \\
&\qquad null \leftarrow visiblePlayer(L) \ \wedge \ movementAttempt(L) \ or \\
&\qquad null \leftarrow movementAttempt(random\_play)
\end{aligned}
$$

The effect of this strategy is that a bot will attempt to follow an enemy if it is in its line of sight. The bot performed slightly better than the single bot using strategy 2.1 but the result was not a significant improvement. The problem was that the bot kept doing one of two things: either its enemy moved quickly out of its line of sight (so it resumed random play) or it sees an enemy on another physical level and in moving towards it the bot itself is forced to go out of line of sight (so it again resumes random play).

Trial 4 : Repeats Trial 3 but with three LCC bots against a single enemy bot. The level of kills is not drastically altered but the bots win more games because the enemy bot's dodging and avoidance

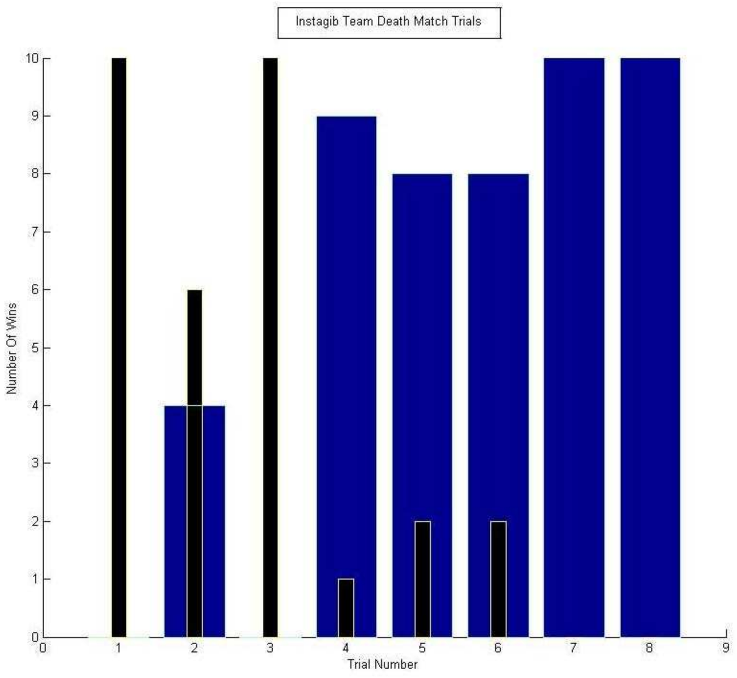


Figure 19: Number of wins for each trial (LCC bots in blue, enemy bots in black)

behaviour is harder to execute when there are multiple enemies all firing at once. Although this strategy has no explicit group coordination, there slight group convergence towards the enemy because of the small size of the level.

Trial 5 : Repeats Trial 3 using three LCC bots against three enemy bots. Given such a simple strategy this result is quite impressive as the bots only lose one less game than when they were playing against only one enemy. More enemy players means more targets to shoot at and less time spent playing entirely randomly.

Trial 6 : In this trial three bots running the following coordination strategy were played against one enemy bot:

$$
\begin{array}{l}
a(team\_hunter, T) :: \\
\quad sawAPlayer(L) \ \Rightarrow \ a(team\_hunter, T1) \leftarrow visiblePlayer(L) \ \wedge \ movementAttempt(L) \ or \\
\quad movementAttempt(L) \leftarrow sawAPlayer(L) \ \Leftarrow \ a(team\_hunter, OT) \ or \\
\quad null \leftarrow movementAttempt(random\_play)
\end{array}
$$

The strategy says that if a bot sees an enemy then they should move towards the enemy and also send out a message to all other bots. Upon receiving this message these bots will then move towards the location where the enemy was last seen (otherwise moving randomly). This allows a basic, communal sense of enemy location to be achieved. Despite this, the bots did not do significantly better than the previous strategy. This was because the bots took the shortest paths to a sighted enemy, and those paths often involved moving through areas which took the bot away from the enemy first and this meant that the bots viewpoint focus would be on the path and not the enemy. If the bot is visible to the enemy on a large section of this path then it will not be trying to defend itself and is therefore vulnerable.

Trial 7 : In this trial three bots running the following strategy was played against one in-built bot, where $strafeAttempt(L)$ is true if the bot attempts to move towards location L while shooting in that direction.

$$\begin{array}{l} a(team\_hunter, T) :: \\ \quad sawAPlayer(L) \Rightarrow a(team\_hunter, T1) \leftarrow visiblePlayer(L) \wedge strafeAttempt(L, L) \ or \\ \quad strafeAttempt(L, L) \leftarrow sawAPlayer(L) \Leftarrow a(team\_hunter, OT) \ or \\ \quad null \leftarrow movementAttempt(random\_play) \end{array}$$

This strategy is similar to that for Trial 6 but strafing is used instead of simple movement commands. Strafing involves looking at a specified target, in our case the enemy. This addresses the problem flaw in coordination in Trial 6. The result is significantly better and, although not hugely different, the average kill counts are swung far enough in the LCC bots' direction that they win every match.

Trial 8 : In this trial three bots running the strategy of Trial 7 were played against three enemy bots. The purpose of this test was to determine how well the strategy scaled up when more enemies were added to the opposing team. The result was that the bots do as well against three enemies as they do against one.

6 Conclusion

In this paper we have described four properties of major concern to those building systems for distributed knowledge sharing (which includes all open, large scale semantic web, grid and multi-agent systems). We have shown how the LCC language can be used as a unifying framework in which to analyse these properties and, for each property, we have shown how traditional forms of simulation and verification can be re-interpreted to gain insights into them. In all cases, generic verification of the general property is impossible so the analytical methods apply to specific forms of engineering intended only to increase the likelyhood that a property is preserved. Such engineering methods are the stock in trade of modern knowledge engineers. By providing a common language and abstract computation model for interactions between knowledge sharing peers we are able to descend into the detail necessary to gain useful knowledge from analysis while retaining a single, abstract system view.

References

[1] A Barker and B Mann. Agent-based scientific workflow composition. In *Astronomical Data Analysis Software and Systems XV*, volume 351, pages 485–488, 2006.

[2] Alan Bundy. Incidence calculus: A mechanism for probabilistic reasoning. *Journal of Automated Reasoning*, 1(3):263–284, 1985.

[3] Li Guo, D Robertson, and Y Chen-Burger. A novel approach for enacting the distributed business workflows using bpel4ws on the multi-agent platform. In *IEEE Conference on E-Business Engineering*, pages 657–664, 2005.

[4] R. J. Hayton, J. M. Bacon, and K. Moody. Access control in an open distributed environment. In *Symposium on Security and Privacy*, pages 3–14, Oakland, CA, 1998. IEEE Computer Society Press.

[5] Sushil Jajodia, Pierangela Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy (SP '97)*, page 31, Washington, DC, USA, 1997. IEEE Computer Society.

[6] Lalana Kagal, Tim Finin, and Anupam Joshi. A policy language for a pervasive computing environment. In *IEEE 4th International Workshop on Policies for Distributed Systems and Networks.* IEEE Computer Society, June 2003.

[7] N. Osman and D. Robertson. Dynamic verification of trust in distributed open systems. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*, Hyderabad, India, 2007.

[8] D Robertson P Besana. Probabilistic dialogue models for dynamic ontology mapping. In *Proceedings of the Second ISWC Workshop on Uncertainty Reasoning for the Semantic Web*, volume 2. CEUR-WS.org, 2006.

[9] D. Robertson, C. Walton, P. Barker, A. Besana, Y. Chen-Burger, F. Hassan, D. Lambert, G. Li, J. McGinnis, N. Osman, A. Bundy, F. McNeill, F. van Harmelen, C. Sierra, and F. Giunchiglia. Models of interaction as a grounding for peer to peer knowledge sharing. In E Chang, T. Dillon, R. Meersman, and K Sycara, editors, *Advances in Web Semantics, vol 1.* Springer-Verlag, LNCS-IFIP, 2007 (to appear).

[10] M P. Singh. Agent communication languages: Rethinking the principles. *Computer*, 31(12):40–47, 1998. Comparing mental vs social agency.