Competence checking for the global e-service society using games

Kostas Stathis[1], George Lekeas[2], and Christos Kloukinas[2]

[1] Department of Computer Science, Royal Holloway, University London, UK
kostas.stathis@cs.rhul.ac.uk
[2] School of Informatics, The City University, London, UK
{g.k.lekeas,c.kloukinas}@soi.city.ac.uk

Abstract. We study the problem of checking the competence of communicative agents operating in a global society in order to receive and offer electronic services. Such a society will be composed of local subsocieties that will often be semi-open, viz., entrance of agents in a semi-open society is conditional to specific admission criteria. Assuming that a candidate agent provides an abstract description of their communicative skills, we present a test that a controller agent could perform in order to decide if a candidate agent should be admitted. We formulate this test by revisiting an existing knowledge representation framework based on games specified as extended logic programs. The resulting framework finds useful application in complex and inter-operable web-services construed as semi-open societies in support of the global vision known as the Semantic Web.

1 Introduction

The vision of the Semantic Web [2] has resulted in a tremendous effort aiming to build an open and distributed infrastructure of ubiquitous and semantic web-services available to both humans and software entities alike. If this effort carries on progressing with the current pace, it is only a matter of time before software components will be in a position to choose from a huge variety of globally available web-services when seeking to achieve their goals, just like humans. The problem then will not be simply how to describe services, publish them, and access them, but also how to organise them, compose them, and enact them, so that any software component can use them in the most effective and flexible manner.

To address the flexible organisation, composition and enactment of web-services, the position of this paper is that current web-services will need to be designed so that they will be part of actions mediated by software agents. Put another way, *agents can offer or receive a service by interacting with other agents*. Provided agents are a suitable abstraction for software components that access or offer web-services [3], the position of this paper goes on further to argue that artificial societies will act as a way of organising the complex interactions involved in composite and heterogeneous services. In this context, *agents can*

offer or receive a service if they are members of an artificial society. The issue then becomes how an agent can be a member of a society [22] and interact with other member agents to receive or offer services.

For autonomous and heterogeneous interactions in artificial agent societies, however, we cannot always assume that (a) we have access to the action-selection strategy of the agent and (b) the protocols available in a society match perfectly with the action-selection strategies of the member agents. In [5] we relaxed (a) so that the action-selection strategy of the agent is kept private but the space of communicative responses is made public [6]. In this way, the agent revealed only the actions it could perform abstractly (e.g. query or refuse in Fig. 1), without giving the conditions under which it would select these acts. Then to address (b) we checked if an agent is *competent*, by checking that the agent is able to *reach* specific states of the interaction (e.g. states s3 and s4 in Fig. 1).

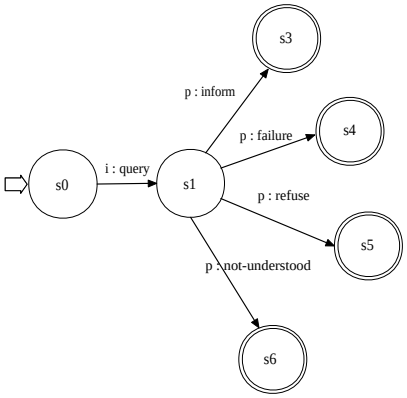

Fig. 1. A simple interaction protocol allowing agents to query other agents about the truth value of a proposition. The protocol starts at state s0 where an agent playing the role of the *initiator* i asks a query, giving rise to state s1. From s1 an agent playing the role of the *participant* p can then reply with: an inform, giving rise to final state s3; a failure, giving rise to state s4; a refuse, giving rise to final state s5; or a not-understood, giving rise to final state s6.

Competence as *reachability* allows us to check whether agents that wish to join a society have the *potential* to terminate the interactions in which they might participate, *provided the other participants allow them to do so.* However, in [5] we did not present the computational part of the competence checking procedure but referred the reader to the games framework in [19]. Here we extend [5] by linking the representation of competence checking using games as the methodology to support the structuring of e-service applications as artificial societies. We look at these issues by concentrating on competence checking of

e-services only, i.e. other related issues such as *trust* or *workflow management* are beyond the scope of this work.

After this introduction, we discuss in Section 2 how to move from the current web-service scenario to one where e-services are mediated by artificial societies, including a social organisation stating how competent agents can become members of societies. In Section 3 we illustrate how interactions in artificial societies can be represented as gaming situations, by providing a concrete computational framework specified in terms of normal logic programs that have a direct Prolog implementation. The resulting computational framework is then extended in Section 4 where we show how to test competence of agents in interactions that require time. Section 5 summarises our contributions, evaluates it, and discusses related and future work.

2 Web-Services, Agents, and the Global E-service Society

2.1 From Web-Services to Agents

A large part of the Semantic Web effort is currently being directed to *web-services*, software systems designed to support machine-to-machine interaction over a network. One of the advantages of the approach is interoperability, i.e., applications written in various programming languages and running on various platforms can use web-services to exchange data over the Internet in a way similar to inter-process communication on a single computer.

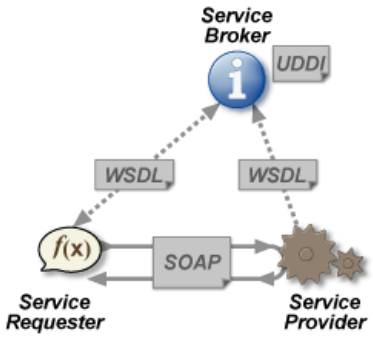

Fig. 2. The diagram, taken from [25], shows how the public interface of a web-service is described using WSDL (*Web-Service Description Language*). Other software components interact with a web-service in a manner prescribed by its interface using messages, which may be enclosed in a SOAP (*Simple Object Access Protocol*) envelope. Such messages are typically conveyed using HTTP, and normally comprise XML in conjunction with other web-related standards. Discovery of a new web-service is achieved via the use of UDDI (*Universal Description, Discovery, and Integration*) protocol.

Fig. 2 depicts the typical service provision context, where a *service requester* identifies how to access a web-service by contacting a *service broker*, who holds information about services and how these can be obtained from *service providers*. One issue of Fig. 2 is that although conceptually the participating components are being thought of as roles of artificial or human agents, the figure focuses on the low-level implementation of the communication between parties, further reducing it to web-based protocol standards for distributed programming. There is, obviously, a conceptual gap between the low-level implementation of distributed components and the high-level organisation of service requesters, providers and brokers, as web-services proliferate day by day.

To fill the conceptual gap of Fig. 2 we use the notion of agents as the extra-layer required for one or more web-services with related functionality to be composed into more complex services. These more complex services will be associated with action descriptions that the agent will be capable to perform, either alone, or through communication with other agents. For example, in this view, the web-service interfaces supporting the functionality of a search engine provider, will be designed as the actions of a search agent that is capable of indexing, searching, and presenting a set of documents as URIs. Under this view, a service requester agent will have to communicate with a broker agent to find the search agent and subsequently ask for any required services. Communication between interacting agents will be governed by communication protocols [13] build on top of on an *Agent Communication Language* (ACL) [17]; [7] presents a way of using ACL for agent-based web-services.

In addition (but unlike [7]), we assume that agents rely upon a logical process that allows them to reason about web-services. Such a process is separate from the way agents invoke web-services using low-level protocols such as SOAP. We achieve this separation by assuming that agents are build with a *mind* and a *body*. The mind of the agent allows us to describe the logical reasoning the agent needs to do, including the planning required to offer a complex service by composing basic services. Agents are competent in providing services, represented in the mind as complex terms. The logical term below:

```
order("Item":string, "Quantity":integer)
```

shows how an agent might represent a more realistic `order` in the context of the protocol defined in Fig. 3. On the other hand, the body situates the mind in the distributed infrastructure of the Semantic Web. Through the body's sensors and effectors the different low-level protocols such as WSDL, UDDI, or SOAP will be used to execute actions and observe the environment. For example, the body will perform an action about an order by translating them in an XML format as shown in the term below.

```
<message name = "order">
<part name = "Item" type="xsd:string"/>
<part name = "Quantity" type="xsd:integer"/>
</message>
```

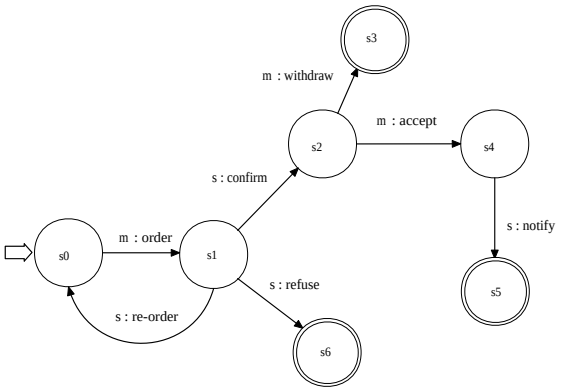

Fig. 3. A protocol where an agent in the role of a *manufacturer* m makes an order in s0, giving rise to situation s1. From s1 an agent in the role of a *supplier* s can reply with: a confirm, stating that the order must be confirmed by m, giving rise to state s2; a refuse, stating that the order cannot be carried out, giving rise to final state s6; and a reorder, stating that the order must be re-specified by m, returning to the initial state s0. If m is asked to confirm in s2, then it may reply either with a withdraw, giving rise to final state s3, or an accept, in which case the state s4 is reached. From s4 the supplier s needs to notify agent m on the details of the transaction, giving rise to final state s5.

This kind of mind-body organisation has already been tested successfully in [20], where the logical actions of agents are translated into physical XML documents that are in turn communicated using the P2P system JXTA [23].

2.2 Requirements for the Global E-service Society

Although agents and the roles they play provide a first-level of semantic organisation of a set of web-services with related functionality, we argue that complex web-services can be best organised at another (higher) level as artificial agent societies. In this view we will use the notion of a *global society* structured in terms of *local sub-societies* as shown in Fig. 4. An agent will belong to a sub-society to start with and use the global society to communicate with agents from other sub-societies. To communicate in the global artificial society agents must be conversant in a global ACL (ACL_G in Fig. 4), possibly different to the local ACLs (such as ACL_K and ACL_N in Fig. 4) used in sub-societies. This choice of allowing different ACLs is not intended to ignore standards, but simply acknowledges heterogeneity, if it exists within an application.

The global society will be open in the sense of [14], while the local sub-societies might be in addition semi-open as in [4]. Members of a local sub-society will be individual agents acting as brokers, service requesters, and service providers, amongst other. To access a web-service within a particular sub-society,

an agent must become a member if the sub-society is semi-open; we use semi-open societies to model the proliferation of web-sites that require registration for example. To join a sub-society we assume that candidate agents must reveal their *service needs*. A candidate agent will also need to make public to the sub-society it wishes to join the *service abilities* it can offer to the society. Service abilities are required so that a society can check whether the candidate agent can participate effectively in the service centric interactions within the society.

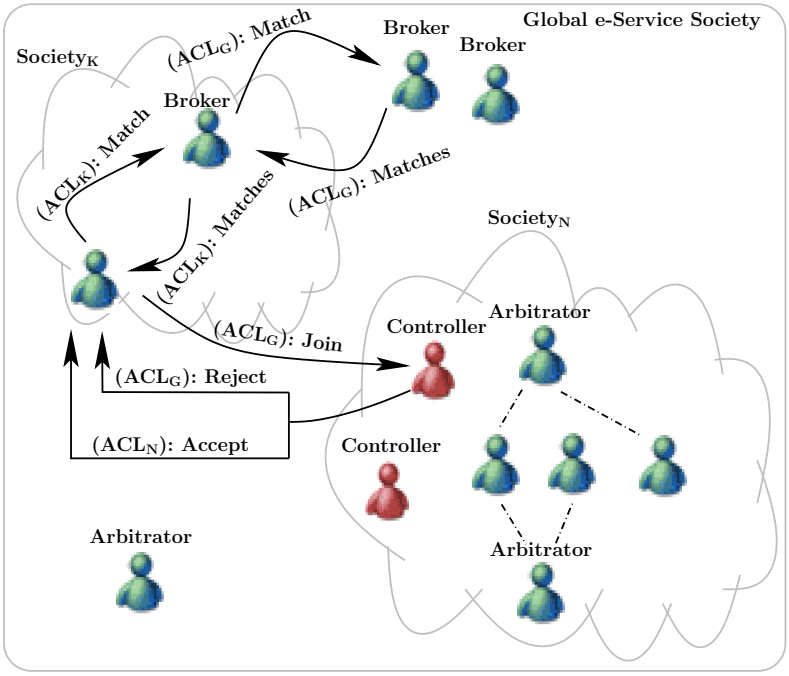

Fig. 4. Agent joining a society

Once the agent is allowed in a sub-society, the agent is given a particular *social position*, implying that the agent will be expected to play one or more *roles* associated with that position. The roles an agent plays condition the agent's participation in interaction *protocols* used in a society, thus regulating the interaction of the agent while receiving or offering a service. According to the protocols available in a society, we assume that a society will have a range of social positions on offer, with certain agents occupying some of these positions already, according to the way the society is organised. Apart from the usual positions encapsulating the roles of Fig.2, we anticipate to need (a) *controllers* to approve/disprove the entry of agents in a society and (b) *arbitrators* to observe

the interactions between parties to enforce the social rules during the provision of a service.

The formal representation of a global society is beyond the scope of this work. Our primary concern is to use this notion informally to contextualise the knowledge representation framework that we propose in the next section, which is the main contribution of this work. This framework aims primarily to help with developing the functionality of controller agents, although from existing work it addresses well known issues with the development of arbitrators as game umpires [19].

3 Competence checking using Games

Following earlier work on the games metaphor [21], we view communicative interactions about web-services within an agent society abstractly as game interactions [19]. More specifically, communicative acts about web-service execution and enactment are made according to protocols which are interpreted as moves made by players of a, possibly, complex game. We do not always look for a winner or a looser, but for the interaction to reach situations with a result, as in dialogue games [18]. We are motivated by communicative interactions that we envisage will play a role in web-services for e-commerce applications, see Fig. 3.

We represent the rules of a game as an normal logic program written in Prolog:

```
game(Situation, Result):-
       terminating(Situation, Result).
game(Situation, Result):-
       \+ terminating(Situation,_),
       valid(Situation, Move),
       effects(Situation, Move, NewSituation),
       game(NewSituation, Result).
```

To formulate a particular game we need to decide how to represent a game situation, its initial and terminating states, how players make valid moves, and how the effects of these moves change the current situation in to the next one until the terminating state is reached. In defining these details we specify what a controller agent needs to reason about how candidate agents can reach potentially terminating situations, by exploring the effects of valid moves for a social protocol. We will further extend this mechanism to plan for basic and complex interactions, involving many agents, to check the competency of such agents.

3.1 Game Situations

We represent game situations by terms of the form:

```
sit(Name, Id, Narrative).
```

Such a term labels a game with a `Name` represented by a constant, a unique `Id` also represented by a constant, and a `Narrative` of moves represented as a list. The `Narrative` can be empty, in which case the term represents the initial situation of a game. The term:

```
sit(order, s0, [])
```

can be thought of as representing the initial situation of the protocol depicted in Fig. 3 represented as a game. Game situations change by players making moves. The term:

```
select(Player, Action)
```

represents the fact that a `Player` has performed an `Action` as his move. For simplicity of presentation we will use here ground terms such as `order` or `confirm` to exemplify the discussion about action terms. Although such representation abstracts away from the content description of these actions, in practice terms will still be ground as we saw with the `order` term in section 2.1:

```
order("Item":string, "Quantity":integer).
```

We are now in a position to deal with what holds in the state of the game as a result of moves made by players. We combine our formulation with the *situation calculus* [11] expressed as a normal logic program:

```
holds(sit(Name, Id, []), F):-
     initially(sit(Name, Id, []), F).
holds(sit(Name, Id, [M | Ms]), F):-
     effect(F, M, sit(Name, Id, Ms)).
holds(sit(Id, [M | Ms]), F):-
     holds(sit(Name, Id, Ms), F),
     \+ abnormal(F, M, sit(Name, Id, Ms)).
```

Given this representation we need to express what holds initially, how effects of moves introduce new fluents, and how fluents that may hold abnormally can be excluded.

3.2 Initial and terminating states

Consider a game where a manufacturer `p1` and a supplier `p2` want to communicate according to the protocol shown in Fig. 3. These roles will need to be specified when the game starts. Then the typical state of such a protocol will need to hold the roles of the players using a `role_of/2` fluent. Another fluent `last_move/1` will also be used to record the last move made. We can express the initial state of this protocol as:

```
initially(sit(order, s0, []), role_of(p1, manufacturer)).
initially(sit(order, s0, []), role_of(p2, supplier)).
```

The absence of `last_move/1` from the initial situation allows our formulation of the situation calculus to capture that this does not hold, using negation as failure. Similarly, we can specify the terminating states of the protocol in Fig. 3 as:

```
terminating(Situation, Situation):-
       Situation = sit(order, Id, N),
       holds(Situation, last_move(select(P, Act))),
       on(Act, [refuse, withdraw, notify]).
```

In other words, in this instance we return as the result the whole of the situation term with all the moves selected so far.

3.3 Valid Moves

Differentiating between valid and invalid moves is of great importance in the analysis of interactive systems as games [18]. For social interactions using agents such as an auction this differentiation will allow the auctioneer to determine which bids are valid and therefore, which bids are eligible for winning the auction [1]. In our games framework, we represent valid moves as:

```
valid(S, Move):- available(S, Move), legal(S, Move).
```

Available moves are all the moves afforded by the state of a game. To represent that order is an available move for the protocol of Fig.3 we write:

```
available(sit(order, Id, N), select(P, order)).
```

The specification of available moves should allow all specified moves to be selected by agents at any state. By adding conditions to `available/2` rules we can check the type preconditions of actions. As selecting an available move in a game does not always imply that this move is legal, we also need to specify legal moves separately. For example, to represent that it is legal to notify only after an accept as in the protocol of Fig. 3, we write:

```
legal(sit(order, Id, N), select(P1, notify)):-
       holds(sit(order, Id, N), last_move(select(P2, accept))),
       holds(sit(order, Id, N), role_of(P1, supplier)).
```

`last_move/1` is what helps the definition of legal moves to ensure that communicative acts are ordered as expected by the protocol.

3.4 Representation of Effects

To represent the effects of a move on the game we distinguish between the effects of that move on the term representing a game situation and how these effects are brought about in the specific state represented by that situation. For example, to represent the effects on the state of the situation we simply extend the narrative of that situation with the move made:

```
effects(sit(Name, Id, Ms), Move, sit(Name, Id, [Move | Ms])).
```

The effects of such a move on the state representing a situation are obtained implicitly by the situation calculus effect and abnormality rules. To give an example of these predicates we consider again the protocol of Fig. 3. We write:

```
effect(last_move(M), M, sit(order, Id, N)).
```

to represent that when we apply a move on the state, it becomes the last move in the next situation. Note that with our formulation of the rules of a game we do not need to check for the preconditions of a move, as we have checked before the effects are carried out that the move is valid.

We also need to specify any abnormal situations where a fluent holds where it should not. For the protocol of Fig. 3, the assertion:

```
abnormal(last_move(M_old), M_new, sit(order, Id, N)).
```

will ensure that after a new move has been made it is abnormal to consider that the last move is the one made previously.

3.5 Competence checking as planning

Given the formulation of games so far we have a way of describing all valid situations that a set of agents can use according to the social rules of a protocol. In [19] we have shown how such rules can be used by an umpire (arbitrator) that checks conformance of the interactions or by a player who wants to play by the rules. However, [19] did not consider competence. To augment the applicability of the approach we view here competence checking as a particular instance of planning using the rules of the game. We will use the following program to plan according to the rules of a game:

```
plan(game(S, R), S, R):-
       achieved(terminating(S, R), S, R).
plan(game(S, R), S, R):-
       \+ terminating(S, _),
       assume(valid(S, M), S, M),
       apply(effects(S, M, NewS), S, M, NewS),
       plan(game(NewS, R), NewS, R).
```

That is, to plan for a game we need to stop when a terminating state has been achieved. Otherwise, in a non-terminating state, we need to assume a valid move, apply the effects of this move to get a new state, and then carry on planning in that new state.

We define `achieved/3` and `apply/4` simply by calling in Prolog the predicates that they take as their first argument (as they are instantiated in the `plan/3` program):

```
achieved(Terminating, Initial, Result):- call(Terminating).

apply(Effects, S, Move, NewS):- call(Effects).
```

To define `assume/3`, however, we need to rely on competence descriptions of players, which correspond to what we referred to in section 2 as the service abilities of agents. To represent such abilities for an agent we will assume rules of the form:

```
competent(Agent, do(Situation, Act)):- Conditions.
```

A controller agent will need to keep rules of this kind to test the competence of candidate agents. The controller must hold such models for all the members in the society too. For example for the protocol of Fig. 3, consider the models describing the competence of players `p1` and `p2`:

```
competent(p1, do(sit(order, Id, N), order)).
competent(p1, do(sit(order, Id, N), accept)).

competent(p2, do(sit(order, Id, N), reorder)).
competent(p2, do(sit(order, Id, N), confirm)).
competent(p2, do(sit(order, Id, N), notify)).
```

We now define:

```
assume(Valid, Situation, select(Player, Act)):-
      call(Valid),
      competent(Player, do(Situation, Act)),
      acceptable(Situation, select(Player, Act)).
```

While planning, this definition allow us to generate a valid move, check that the agent is competent of performing it, and finally check that a move is acceptable. The definition of `acceptable/2` joins the assumed move with the rest of the narrative describing the current situation to filter unwanted loops. For the protocol of Fig. 3 such a loop is described by the sequence:

```
[select(A, order), select(B, reorder)]
```

which is allowed to be repeated only once. The implementation of acceptable moves for this example is not included here as it trivially checks for specific unwanted sub-lists of a list. We are now in a position to ask:

```
?- plan(Game, sit(order, s0, []), Result)
```

and get as part of the solution process all the valid states that can be planned for using the description of the protocol and the descriptions of the competence for individual players, with loops allowed only once, if they exist. What a controller agent can then do with the results is application specific.

4 Competence checking in Timed Games

Combining our games framework with the normal logic programming formulation of the situation calculus allowed us to specify protocol-based interactions and test for reachability of all the states of the protocol via planning. However, in many occasions social protocols do not assume strict turn-taking in that moves of players can occur at the same time. An example of such a protocol is that of an English auction, as shown in Fig. 5.

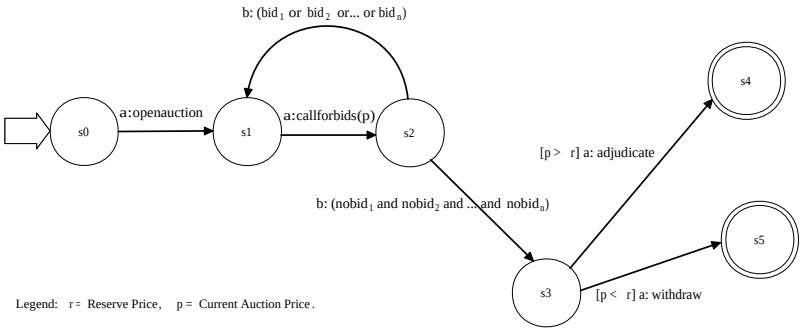

Fig. 5. The English auction protocol allowing an agent with the role of an auctioneer `a` and a set of agents with the role of bidder `b` to interact for the sale of a good. The auctioneer starts the auction and the calls for bids at a specific price. One or more bidders bid, in which case the auctioneer calls for new bids until no more bids are offered. At that point the auctioneer either adjudicates the good to the highest bidder or withdraws the good if the reserve price is not met.

To allow for protocols of the kind describe in the above figure we introduce timed games, that is, games whose moves have also a representation of the time in which they happened.

4.1 Timed Games in the Event Calculus

In trying to formulate timed games we introduce timed situations of the form:

```
sit(Name, Id, Time, Narrative).
```

One difference from our earlier representation is that now we need to keep the current `Time` in the situation term. In addition, a narrative in timed games is represented in terms of *episodes*, that is collections of moves that can validly happen at the same time in a situation. We express episodes as:

```
at([select(Player1, Act1), ..., select(PlayerN, ActN)], T).
```

Using this representation, the term `at(T, [])` means that nothing happened at time T.

To reason about timed game situations, we use the *simple version* [15] of the *event calculus* [8] instead of the situation calculus, suitably adapted for our purposes as follows:

```
holds(sit(N,Id,Tn,Nn), P):-
        0 =< Tn,
        initially(sit(N,Id,Ti,Ni), P),
        \+ clipped(P, sit(N,Id,Ti,Ni), sit(N,Id,Tn,Nn)).
holds(sit(N,Id,Tn,Nn), P):-
        happens(E, Ti, Ni, Nn),
        Ti < Tn,
        initiates(E, P, sit(N,Id,Ti,Ni)),
        \+ clipped(P, sit(N,Id,Ti,Ni), sit(N,Id,Tn,Nn)).

clipped(P, sit(N,Id,Ti,Ni), sit(N,Id,Tn,Nn)):-
        happens(Estar, Tj, Nj, Nn),
        Ti < Tj, Tj < Tn,
        terminates(Estar, P, sit(N,Id,Tj,Nj)).
```

The important difference from the normal event calculus formulations is that narratives are held as lists in situation terms rather than as assertions in the knowledge base, in the spirit of the situation calculus. In this context, our representation of an event happening is re-specified as:

```
happens(E, Tn, [at(En, Tn)|Sn], [at(En, Tn)|Sn]):-
        member(E, En).
happens(E, Ti, [at(Ei,Ti)|Si], [at(En, Tn)|Sn]):-
        happens(E, Ti, [at(Ei,Ti)|Si], Sn).
happens(at(En,Tn), Tn, [at(En,Tn)|Sn], [at(En, Tn)|Sn]).
happens(at(Ei,Ti), Ti, [at(Ei,Ti)|Si], [at(En, Tn)|Sn]):-
        happens(at(Ei, Ti), Ti, [at(Ei,Ti)|Si], Sn).
```

The first two rules deal with individual events as in the simple event calculus, with the difference that now we need additional parameters to keep the narrative at intermediate times. Unlike the simple event calculus however, our formulation also requires additional rules (the last two) to deal with episodes that have happened in the narrative; like the events they contain, they too can affect the state of the game.

One implication of the use of episodes is that we need to change the way we update the narrative in a timed game. We write:

```
effects(sit(N,Id,T,Es), at(Ms, T), sit(N,Id,NewT,[at(Ms,T)|Es])):-
        T > 0,  NewT is T + 1.
```

The above definition makes the assumption that new episodes last for one unit of time. The rest of the generic representation for game remains the same, the only parts that change are the domain specific details. We give an example next.

4.2 Formulating an English Auction

To exemplify timed games we present briefly parts of our formulation for an auction as shown in Fig. 5. For simplicity, we will assume that there are two bidders and an auctioneer, and that in order to check the game we only need the last set of moves captured in the fluent (`last_moves/1`). We will represent the initial state as before, but now we will need to also specify the initial time, which we will assume it is 0. This gives rise to the initial state:

```
initially(sit(auction, s0,0,[]), role_of(p1, auctioneer)).
initially(sit(auction, s0,0,[]), role_of(p2, bidder)).
initially(sit(auction, s0,0,[]), role_of(p3, bidder)).
```

The terminating conditions are specified with holds axioms using the simple version of the event calculus presented in the previous section. For example, to define termination in the auction we write:

```
terminating(sit(auction,Id,T,N), sit(auction, Id, T, N)):-
        holds(sit(auction,Id,T,N), last_moves([select(P,X)])),
        member(X, [adjudicate,withdraw]).
```

The valid moves are specified as before, including available and legal moves, however now these need to be specific to the moves of the auction. For example, to specify a legal bid we write:

```
legal(sit(auction,Id,T,N), select(Player1, bid)):-
        holds(sit(auction,Id,T,N),role_of(Player1,bidder)),
        holds(sit(auction,Id,T,N),last_moves([select(Player2,cfp)])),
        holds(sit(auction,Id,T,N),role_of(Player2,auctioneer)).
```

The only aspect that really changes is the representation of effects, which are now expressed in terms of `initiates/3` and `terminates/3` instead of `effect/3` and `abnormal/3`.

```
initiates(at(Es, T),  last_moves(Es), sit(auction,Id,T,Ns)).

terminates(at(Es, T), last_moves(Old_M), sit(auction,Id,T,Ns)).
```

Notice that in this particular example `initiates/3` and `terminates/3` rules are written only for episodes, however, in general, these need to be specified also for individual events.

4.3 Competence checking of an English Auction

To check the competence of a set of players for timed games we are going to assume, as before, that we have a set of statements regarding the competencies of individual players and the `plan/4` program. The main aspect that changes in timed games, however, is that instead of generating individual moves we need to generate individual episodes:

```
assume(Valid, Sit, at(Moves, T)):-
     Sit = sit(N,Id,T,Es),
     Valid = valid(Sit, select(P, M)),
     findall(M, (call(Valid, competent(P, do(Sit, M))), All),
     sublist(Moves, All),
     acceptable(Sit, at(Moves, T)).
```

In other words, we need to change our definition of `assume/3` to deal with episodes, so that we get all the valid and acceptable subset of moves in the protocol. Running the query:

```
?- plan(Game, sit(auction, s0, 1, []), Result)
```

we will be in a position to find all the reachable states of the protocol, according to the description of the rules, and the competence of the players.

5 Concluding remarks

We have investigated the issue of competence checking for agents operating in a global artificial society whose purpose is to organise complex services. Assuming that a candidate agent provides an abstract description of their communicative competence, we have formulated a test that a controller agent can perform to decide if the candidate agent should join a sub-society of the global society. We have formulated this test by revisiting an existing knowledge-based framework based on games represented in extensive form. Although [22, 5, 9] have motivated our framework, we have found no other related work that links agent competency with artificial societies using games.

In evaluating our approach we see that our formulation can integrate the situation and the event calculi according to the competence checking problem at hand. In this context we inherit from our original formulation of games the notion of *compound games*, viz., games built from *active* sub-games [18], thus allowing quite complex interactions to be checked for competency. Also, by using normal logic programs our approach can be implemented directly in Prolog, unlike other approaches that need to extend the proof-procedure e.g. agents based on abduction [22].

The current formulation of games and, as a result, the competence checking presented has the potential to build upon the methodology developed in [18]. One aspect of this is that it treats valid acts as an abstraction for different specification approaches of social action, as they may be required by different applications. We have for example assumed that valid acts must be available and legal. Nevertheless, not all applications need to be presented in this way. For example, in [1] valid acts are treated in a way that relies on a more elaborate representation of concepts such as those of obligation and permission. Investigation of these aspects will allow us to compare our framework with existing approaches that model web-services, e.g. see [12], but with an artificial societies approach.

By investigating how to best check the competency of agents in artificial societies for e-services we have identified the need to incorporate into our framework a mechanism that ensures that agents are not simply competent according to the acts of a protocol but also according to the expected order of acts described in it. In parallel, we also need to deal with the re-computation introduced from the use of event and situation calculi in more complex domains to the examples used here. An immediate remedy will be to run our games framework on a Prolog system that supports tabling [24], such XSB Prolog. How tabled execution compares with approaches based on model checking [10] and satisfiability [16] is another direction that we wish to investigate in this context.

Acknowledgements

We would like to thank the anonymous referees for their comments on a previous version of this paper. The first and third authors were partially supported by the EU IST6 *ArguGRID* and *SeCSE* projects respectively.

References

1. A. Artikis, J. Pitt, and M. Sergot. Animated specifications of computational societies. In C. Castelfranchi and W. Lewis Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2002), Part III*, pages 1053–1061, Bologna, Italy, July 15–19 2002. ACM Press.
2. T. Burners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 284(5), May 2001.
3. V. Curcin, M. Ghanem, Y. Guo, K. Stathis, and F. Toni. Building next generation Service-Oriented Architectures using Argumentation Agents. In A. Polze and R. Kowalczyk, editors, *3rd International Conference on Grid Service Engineering and Management*, pages 249 – 263, Germany, Sep 2006.
4. P. Davidsson. Categories of artificial societies. In P. Petta A. Omicini and R. Tolksdorf, editors, *Engineering Societies in the Agents World II*, pages 1–9, Prague, Czech Republic, 2001.
5. U. Endriss, W. Lue, N. Maudet, and K. Stathis. Competent agents and customising protocols. In A. Omicini, P. Petta, and J. Pitt, editors, *Proceedings of the 4th International Workshop Engineering Societies in the Agent World (ESAW-2003)*, volume 3071 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 168–181. Springer-Verlag, 2004.
6. U. Endriss, N. Maudet, F. Sadri, and F. Toni. Protocol conformance for logic-based agents. In G. Gottlob and T. Walsh, editors, *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico (IJCAI-03)*. Morgan Kaufmann Publishers, August 2003.
7. R. Kowalczyk J. Yan, Y. Yang and X. T. Nguyen. A service workflow management framework based on peer-to-peer and agent technologies. In *Proc. of International Workshop on Grid and Peer-to-Peer based Workflows*, Melbourne, Australia, Sep. 2005.

8. R. A. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–95, 1986.
9. G. K. Lekeas and K. Stathis. Agents acquiring Resources through Social Positions: An Activity-based Approach. In O. de Bruijn and K. Stathis, editors, *Proceedings of the 1st International Workshop on Socio-Cognitive Grids*, Santorini, Greece, June 2003.
10. A. Lomuscio and F. Raimondi. Model checking knowledge, strategies, and games in multi-agent systems. In *Proceedings of the 5th International Conference on Autonomous Agents and Multi-Agent systems (AAMAS06)*. ACM Press, 2006.
11. J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. American Elsevier, New York, 1969.
12. S. McIlraith, T. Cao Son, and H. Zeng. Semantic web services. *IEEE Intelligent Systems*, 16(2):46–53, 2001.
13. J. Pitt and A. Mamdani. A Protocol-based Semantics for an Agent Communication Language. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, Stockholm, Sweden (IJCAI-99)*, pages 486–491. Morgan Kaufmann Publishers, 1999.
14. J. V. Pitt. The open agent society as a platform for the user-friendly information society. *AI & Society*, 19(2):123–158, 2005.
15. M. Shanahan. The event calculus explained. In M.Veloso M.J.Wooldridge, editor, *Lecture Notes in Artificial Intelligence*, pages 409–30. Springer, 1999.
16. M. Shanahan and M. Witkowski. Event Calculus Planning Through Satisfiability. *Journal of Logic and Computation*, 14:731–745, 2004.
17. M. P. Singh. Agent communication languages: Rethinking the principles. In *Communication in Multiagent Systems*, pages 37–50, 2003.
18. K. Stathis. *Game–Based Development of Interactive Systems.* PhD thesis, Department of Computing, Imperial College London, Nov 1996.
19. K. Stathis. A Game-based Architecture for developing Interactive Components in Computational Logic. *Functional and Logic Programming, Special Issue on Logical Formalisms for Program Composition*, 2000(1), March 2000.
20. K. Stathis, A. Kakas, W. Lu, N. Demetriou, U. Endriss, and A. Bracciali. PROSOCS: a platform for programming software agents in computational logic. In J. Müller and P. Petta, editors, *Proceedings of the Fourth International Symposium "From Agent Theory to Agent Implementation"*, Vienna, Austria, April 13-16 2004.
21. K. Stathis and M. J. Sergot. Games as a Metaphor for Interactive Systems. In M. A. Sasse, R.J. Cunningham, and R. L. Winder, editors, *People and Computers XI (Proceedings of HCI'96)*, BCS Conference Series, pages 19–33, London, UK, August 1996. Springer-Verlag.
22. F. Toni and K. Stathis. Access-as-you-need: a computational logic framework for flexible resource access in artificial societies. In *Proceedings of the Third International Workshop on Engineering Societies in the Agents World (ESAW'02)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 2002.
23. B. Traversat, M. Abdelaziz, D. Doolin, M. Duigou, J. C. Hugly, and E. Pouyoul. Project JXTA-C:Enabling a Web of Things. In *Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS'03)*, pages 282–287. IEEE Press, January 2003.
24. D. S. Warren. Memoing for logic programs. *Commun. ACM*, 35(3):93–111, 1992.
25. Web-services. Home Page: http://en.wikipedia.org/wiki/Web_services.