

# A Design Method for Logic Language-oriented MAS that use Web Services

Steve Goschnick and Sandrine Balbo

University of Melbourne

stevenbg@unimelb.edu.au, sandrine@unimelb.edu.au,

## Abstract

This paper details a new design method suitable for logic programming in general, but particularly within multi-agent systems (MAS). The DigitalFriend is an example of a user-friendly MAS aimed at the end-user as developer, with a built-in logic language interpreter. It uses fragments of logic code as dynamic glue, bringing together numerous sub-agents that may exist as Web services, to form new services. A gap in current logic language design methods is identified. The well-established techniques of *Task Analysis* and entity relation *Normalization* are brought together in a new Design Method (called TANDEM) to address the gap. This method is demonstrated with an example application - the ‘movie-cinema problem’ - built using the DigitalFriend.

## 1. Introduction

Multi-Agent Systems (MAS) are increasingly being used to integrate a number of distributed, individually published web services, into new more complex and useful services. Each individual participating service is wrapped as an individual agent. By combining them with other such agents and agents of other types, a new useful more sophisticated application is formed, often unforeseen by the publishers of the individual Web services. In industry these new applications are now known as *Mash-ups*.

One technology that has been designed with such functionality in mind, is the DigitalFriend [6], a MAS-based on the ShadowBoard agent architecture [4]. It has a built-in Prolog-like language called CoLoG. Logic programs within the DigitalFriend are dynamically constructed as needed from constituent sub-agents during the systems normal operation, and also discarded soon after running. Some of the sub-agents have sets of *predicates* and *rules* to choose from, most have a set of different *goals* they can call upon to pursue, others retrieve *ground terms* (i.e. literal information) dynamically, often from remote web services. The order and firing of sub-agents, and hence the dynamic construction of a given logic program, is analogous to the order and firing of sub-plans in a BDI agent architecture.

Devising the logic predicates and rules, which are distributed amongst the sub-agents, then becomes a primary activity when building a given application using such a logic language-oriented MAS.

This paper details a new *design method* suitable for developers using such multi-agent systems, to wrap web services and other agents, with a complete example. The

methodology draws upon both *Task Analysis* and data *Normalization* (from entity relation modeling) in a new *DEsign Method* named *TANDEM*, introduced in a short paper here [5]. The method is demonstrated in the design of a *movie-cinema* location application, typical of a modern application scenario that gathers constituent components from distributed web services. An overview of the implementation using the DigitalFriend system is then given. The method has broader applicability to logic programming (LP) in general.

## 2. The Need for a Design Method

The DigitalFriend is aimed at development by the end-user, at about the level of a spreadsheet or desktop database user. It requires these users to formulate small logic programs, which in turn generates the need for a *design method* to devise good logic programs, in a straight-forward manner. Sterling [8] gives a definition of a good logic program as one that is: ‘*declarative, easily understood, and able to efficiently solve the problem at hand*’.

In their well-regarded text on the logic language Prolog, within the chapter on development, Sterling and Shapiro [9:237] reflect that the design of a good logic program is generally achieved through rapid prototyping, evolving it through rewriting and extension. Some years later, Sterling applied the concept of *patterns* to logic programs [8], and identified two classes of patterns, namely *skeletons* and *techniques* – the first class is to do with program *control flow*, and the second targets generalized, method-like operations, that have a wide range of applicability.

He notes ‘*Despite attractive features, Prolog has not been widely adopted within software engineering. Standard development practices have not been adapted to Prolog. A major area of weakness is design... Nothing analogous to design techniques, such as structure analysis for procedural or object oriented design for object-oriented languages, have been developed for logic languages.*’ [8:10]

He then includes the two design patterns of *skeletons* and *techniques* into a method of design called *stepwise enhancement*, which is a more refined and formal specification of the rapid prototyping approach in the earlier mentioned book. A part of the formalism involves ‘*listing predicates and their terms*’, but no specific method is advocated for identifying the correct, best or most useful predicates nor their appropriate terms, beyond the

'*experience of the programmer*'. Predicate refinement is via iteration alone, in that design approach.

While the pattern-oriented approach is useful to regular or advanced logic programmers and analysts writing large logic programs, there is little benefit in the method to the logic language novice, simply devising good predicates.

### 3. The TANDEM Design Method

We present a method for deriving well-formed logic predicates and rules, for an MAS application. It combines a Task Analysis [1,2] with the Normalization technique from entity relation (ER) analysis [3]. Given the significant conceptual correlation between database *relations* and *predicates* in logic languages, and between a relation's *attributes* and a predicate's *terms*, we incorporated normalization within TANDEM. The analysis begins with a collection of all possible *entities* that a requirements gathering exercise uncovers. Entities are usually a specialization of either a: Person (e.g. *patient*), Place (e.g. *cinema*), Object (e.g. *vehicle*), Event (e.g. *registration*) or Concept (e.g. *account*). In the actual system implementation, information represented by these entities identified in the analysis, will often be sourced from a number of unrelated web services distributed across the Internet - and communicated by agent communication acts. Nonetheless, the initial ER diagram is modeled traditionally as if it were going to be implemented in one single system.

#### 3.1 The Movie-Cinema Problem

We present the methodology by way of an example application, one typical of year-long projects developed by software engineering students at the University of Melbourne, centered on fulfilling an industry need. Such projects often involve Internet resources and complex problems, which can usually benefit from recent research and development advances in software agents and MAS.

**The problem:** The client, a movie cinema operator, wants a system accessible by movie-going customers via the Internet, one which will allow users of smart hand-held devices, to search for specific movie details, ultimately leading to the booking of cinema tickets.

- The client has a number of cinemas, with names that reflect their locations e.g. 'City Centre' is the name of the cinema within the city itself, while 'Chadstone' is the name of the cinema at the Chadstone Shopping Centre. They have many such cinemas, but never more than one per location.
- When the user has a smart device capable of determining their location, inform them of the closest cinemas.
- After the user has chosen a cinema, they should be shown the movies currently playing there. When they have chosen a movie, they should be shown which cinemas are playing it.
- Each cinema has a number of theatres in which movies are viewed. There are three types of theatre, namely: Standard, Luxury, and Art-house (where they show less commercial

movies). A selected cinema may for example have: 8 standard theatres, 2 art-house theatres and 1 luxury theatre, but these numbers vary from cinema to cinema. Each theatre has a number of seats, which can be selected when a customer buys a ticket online. A seat is identified by its seat row and its seat number within that row.

- The client wants to make available details on the movies that are currently playing, including: title, rating, director, length, main cast members (i.e. known stars), rating (i.e. G, PG, M, MA, R), genre (e.g. drama, horror, comedy), official movie website, release date. If an alternative service can be found which can provide this information dynamically, then that is a preferred option, so as to fast-track the project.
- A movie can be screening (played) in one or more theatres consecutively. A movie will usually be played during one or more sessions within a given day. Details about each session include: day, start-time, number-of-features (i.e. One session can screen either one movie, or two – called a double-feature). Operationally, the system never needs to hold more than seven days worth of *sessions* – i.e. the movies that are screening from 'today', forward one week.
- Both *available* and *allocated* seats should be displayed on a graphic image of the theatre plan, allowing the user to select seats during a booking, as required.
- Ticket types are either: *Adult*, *Student*, *Child*, *Pensioner* or *Senior Citizen*. Each ticket-type has a different cost. However, the cost of a given type remains the same across all movies at all cinemas that the client owns and operates.

Once a movie-going customer purchases a ticket, full ticket details are to be provided, including: Cinema, Movie title, day, start-time, ticket-type, cost, seat row and seat number.

#### 3.2 Phase 1 - Applying Task Analysis

Task Analysis identifies a user's goals and the procedure or steps needed to achieve them. It includes the process of analyzing the structure of a task and decomposing the task into related sub-tasks. Some of the tasks identified using a task analysis and are listed below as *verb-first* directives:

##### Task 1 - Movie-oriented search:

- *Find* all recently released movies.
- *Find* all movies within a particular genre.
- *Find* which movies feature a specific actor.
- *Find* all movies by a specific director.

##### Task 2 - Cinema-oriented search:

- *Determine* recent movies playing at a local cinema.
- *Determine* which cinemas are playing a specific movie.
- *Find* what ticket types are available at a specific theatre.
- *Find* the cost of a specific ticket type.

##### Task 3 - Location-oriented search:

- *Find* all cinemas within a given radius of the user's position.
- *Find* all cinemas within the locality of a postcode.

- Calculate the distance to the nearest cinema where a specific movie is playing.

**Task 4 - Seat-oriented search and ticket allocation:**

- Select unallocated seats for movie and theatre of choice.
- Select ticket type/s.
- Enter name and payment method.
- Allocate the booked seats.

These four groups of tasks and sub-tasks - represent the first step in a *hierarchical task modeling* exercise [1].

This task modeling phase could employ any of a considerable number of *task analysis notations*, as described by Balbo et al [2], who review six well-known and widely used notations. However, we do not need to proceed further with task analysis, as the logic rules we derive from the method, can answer the many diverse tasks in our problem above, and more, discussed in Section 4.

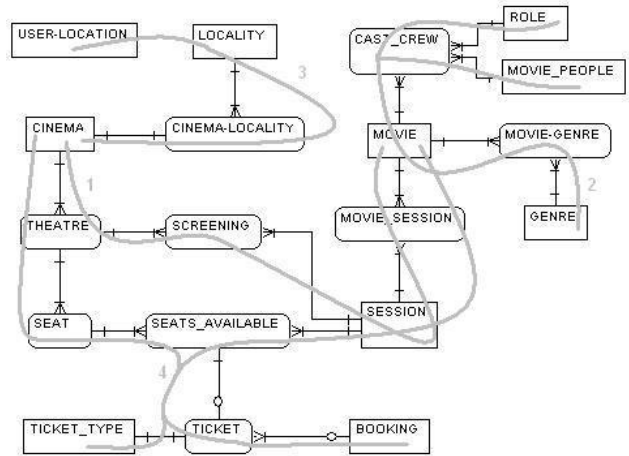
**3.3 Phase 2 - Applying ER Normalization**

In database applications a technique for designing new systems that is widely used and widely taught in both Information Systems and Computer Science, is entity relation (ER) modeling [3]. And further, a technique that is applied to an ER model to get it into an *ideal form* - a form equally suited to all manner of SQL-based queries put to the implementation, and one that is most flexible with regard to future change requirements, is called *Normalization* [3].

We apply this well-known technique as a step within the TANDEM method - and since it is well-known, the process of normalization is not outlined here. We applied it to the movie-cinema problem above, and arrived at the normalized entity relation (ER) model in Figure 1.

Then, in order to divide the ER diagram into appropriate sub-areas, each of which will be represented by a sub-agent, we identify *logic flow lines*, as also depicted in Figure 1. These lines are mapped by taking the four high level tasks identified in the Task Analysis, and then identifying which of the entities in the ER diagram are involved in each task - *without breaking the line*.

The reason for not breaking the line is that logic languages, during the process of *unification* [9], dynamically join each predicate expressed in a rule, wherever there is a commonly named attribute between them – in order to satisfy a query. The major difference with respect to an SQL relational database management system (RDBMS), is that the RDBMS only joins entities (i.e. predicates), on common attributes, *as specified by the SQL*. A logic language makes all possible joins automatically, often building a huge *space state* (a *flattened table*, in RDBMS terms). The relationships between entities in Fig. 1 (the straight lines), each infer a common attribute between the pair of entities at both ends of a given line. The curved lines in Fig. 1, identify just those entities we group together in a sub-



**Fig.1 – Normalized Entity Relation Diagram Modelling the Cinema Problem, with four Logic Flow Lines**

model. Hence, *to break the line* – meaning to have two entities that have no relationship line between them - would prevent a logic program from successfully answering a query via unification.

**3.4 Phase3 - Deriving Well-formed Logic Rules**

In a logic program, the predicate at the *head-of-the-rule* is a template for many of the queries that can be answered by the whole program. The total set of such rules, gives all templates for all queries that can be answered by the system. In the MAS implementation, these rules and their associated logic, is distributed between the sub-agents.

In our design solution five rules are derived, explained in the next four sub-sections, with the following five *head-of-the-rule* predicates:

**cinemaAssistant**(CinemaName, Location, TheatreType, MovieTitle, GenreName, Rating, Date, StartTime)

**movieAssistant**(MovieTitle, Rating, GenreName, RunningTime, DvdRelease, RoleName, FirstName, Surname)

**localityAssistant**(Distance, Radius, Location, TownSuburb, Postcode, CinemaName)

**seatSelection**(CinemaName, Location, TheatreID, ImageMap, MovieTitle, Date, StartTime, RowNumber, SeatNumber, CoordX, CoordY, allocated)

**ticketAssistant**(CinemaID, TheatreID, Date, StartTime, RowNumber, SeatNumber, TicketNumber, TicketType, Cost, BookingID, FirstName, LastName, PaymentMethod, PaidStatus)

The *terms* within the brackets of the predicate, all begin with a capital letter, to indicate that they are variables. A query is constructed by replacing one or more of those variables with a specific *instance*, for example, the query:

*What movies feature Sharon Stone as a star?*

Would be specified to the CoLoG program as follows:

movieAssistant(**MovieTitle, Rating, GenreName, RunningTime, DvdRelease, star, sharon, stone**)?

It can also be specified with under-scores in place of the variable names, leaving only the need to supply the literals (individual terms), as follows:

movieAssistant( \_, \_, \_, \_, \_, *star, sharon, stone*)?

In the TANDEM method, we first decide on all the head-of-rule predicates that will be necessary to answer all of the tasks extracted via the Task Analysis. The four groupings of *primary tasks*, told us that we needed the four predicates as detailed above. We simply settled upon the four predicate names - *cinemaAssistant*, *movieAssistant*, *localityAssistant* and *ticketAssistant* - as meaningful names that were utilized within the MAS implementation, outlined in Section 4.

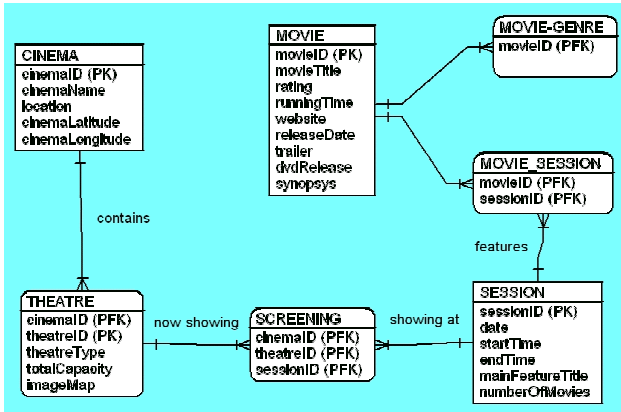


Fig.2 – An ER diagram detailing attributes for entities associated with the *cinemaAssistant* rule.

### 3.4.1 Deriving the Cinema Assistant Rule

To identify the actual *terms* needed by each predicate, we use a more detailed diagram than the ER diagram depicted in Figure 1. We could have chosen to depict all sub-area entity *attributes* in a more detailed style of ER diagram as seen in Figure 2. However, in this paper, we have chosen to depict attributes for each of the sub-areas identified by a *logic flow line* in Figure 1, as UML Class diagrams – assuming that more application developers are likely to be familiar with this model notation. E.g. The equivalent UML class diagram for the ER diagram expressed in Figure 2, can be seen in Figure 3. Both figures depict the sub-area entities involved in the *cinemaAssistant* rule. (From here on down, we use only the more familiar UML class diagram).

The constituent *terms* within each predicate rule are then determined via the atomic tasks that made up the high level task for which the predicate was instigated.

The second stage of this phase involves identifying the best configuration of predicates on the *right hand side* of each rule in turn:

#### Rule 1:

Head of the Rule:

**cinemaAssistant**(*CinemaName*, *Location*, *TheatreType*, *MovieTitle*, *GenreName*, *Rating*, *Date*, *StartTime*) ←

Requires the following Predicates (from the normalised ER diagram) on the right hand side of the *implies* (←):

**cinema**(*CinemaID*, *CinemaName*, *Location*, *CinemaLatitude*, *CinemaLongitude*),

**theatre**(*CinemaID*, *TheatreID*, *TheatreType*, *TotalCapacity*, *ImageMap*),

**session**(*SessionID*, *Date*, *StartTime*, *EndTime*, *MainFeatureTitle*, *NumberOfMovies*),

**movie**(*MovieID*, *MovieTitle*, *Rating*, *RunningTime*, *Website*, *ReleaseDate*, *Trailer*, *DvdRelease*, *Synopsis*),

**movieGenre**(*MovieID*, *GenreName*),

**screening**(*CinemaID*, *TheatreID*, *sessionID*),

**movieSession**(*MovieID*, *SessionID*);

Note 1: From here onwards, these predicates are called the *body of the rule*.

Note 2: The commas between predicates are logical *conjunctions*.

These *required* predicates making up the *body of the rule*, are identified via their member terms within the entity/class. i.e. Each class in Figure 3 that has a member *attribute* which corresponds with a *term* within the head-of-the-rule *predicate*, is included as *rule-body* predicate. These matching member attributes are highlighted in **bold-italic** font above.

Note that the last two predicates have no such matching terms in the head-of-rule predicate, but are there as *connective* predicates, between the other necessary predicates. That is, they provide a path - in the *logic flow line* - for the search-space constructed within an executing logic program, to include all necessary *terms*, in order to satisfy the rule. In ER models such connective relations are called *associative relations*. In UML class models, they are called *association classes*.

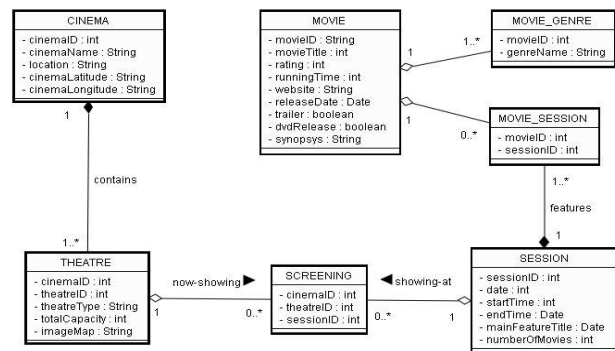


Fig.3 – A UML Class diagram detailing attributes for entities associated with the *cinemaAssistant* rule.

### 3.4.2 Deriving the Movie Assistant Rule

Figure 4 depicts the sub-area entities involved in the *movieAssistant* rule. Again, all of the appropriate entities/classes have been identified for this sub-area, by following the appropriate *logic flow line* in Figure 1. The

entities are all to do with movies, with nothing to do with cinemas or locations. A user will use the agent encompassing this rule, to enquire about a movie, whether or not they ever intended to go and see it in a cinema. I.e. It will enable a useful standalone agent.

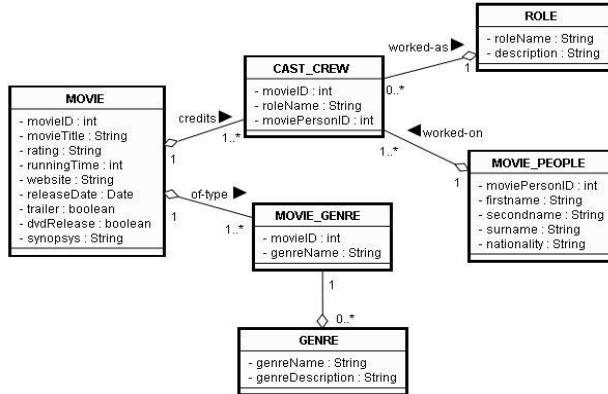


Fig.4 – A UML Class diagram detailing attributes for entities associated with the movieAssistant rule.

**Rule 2:**

Head-of-the-rule:

**movieAssistant**(*MovieTitle, Rating, GenreName, RunningTime, DvdRelease, RoleName, FirstName, Surname*) ←

Requires the following predicates in the rule body, via the normalized entities, represented with their attributes in Figure 4:

**movie**(*MovieID, MovieTitle, Rating, RunningTime, Website, ReleaseDate, Trailer, DvdRelease, Synopsis*),

**moviePeople**(*MoviePersonID, Firstname, Secondname, Surname, Nationality*),

**movieGenre**(*MovieID, GenreName*),

**genre**(*GenreName, GenreDescription*),

**castCrew**(*MoviePersonID, MovieID, RoleName*),

**role**(*RoleName, Description*);

**3.4.3 Deriving the Location Assistant Rule**

Figure 5 depicts the sub-area entities involved in the *localityAssistant* rule. As before the appropriate logic flow line identifies the necessary entities for the sub-area.

The rule is then determined accordingly.

**Rule 3:**

Head-of-the-rule:

**localityAssistant**(*Distance, Radius, Location, TownSuburb, Postcode, CinemaName*) ←

Requiring the following Predicates in the rule body, via normalized ER diagram:

**cinema**(*CinemaID, CinemaName, Location, CinemaLatitude, CinemaLongitude*),

**locality**(*LocalityID, TownSuburb, PostCode, StateCode, CountryCode, CentralLatitude, CentralLongitude*),

**cinemaLocality**(*cinemaID, localityID*),

**userPosition**(*CurrentLatitude, CurrentLongitude*);

Plus the following predicate and a little CoLoG logic:

**distanceBetween**(*Distance, Longitude1, Latitude1, Longitude2, Latitude2*),

**#**(*CurrentLongitude = Longitude1*),

**#**(*CurrentLatitude = Latitude1*),

**#**(*CinemaLongitude = Longitude2*),

**#**(*CinemaLatitude = Latitude2*),

**#**(*Distance <= Radius*);

The *distanceBetween* predicate is not in the ER diagram, so where does it come from in the method? Via the need for *distance* in the rule head, and by observing that we have

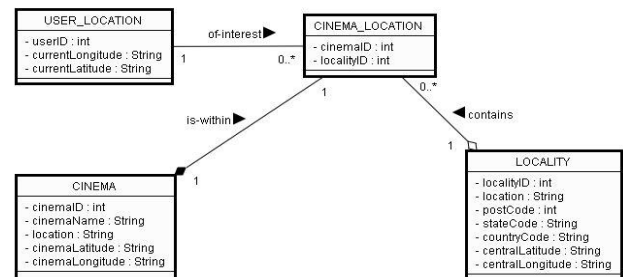


Fig.5 – A UML Class diagram detailing attributes for entities associated with the localityAssistant rule.

terms for *longitude* and *latitude* in both the *cinema* and the *user-location* relations in Figure 1. In the implementation it will need to be provided with some extra logic, or via a wrapped web service that is able to calculate the distance between two global coordinates.

The four lines of code below *distanceBetween* simply assign the appropriate terms from *cinema* and *userPosition*, as the two global coordinates that will be used to calculate a *distance*.

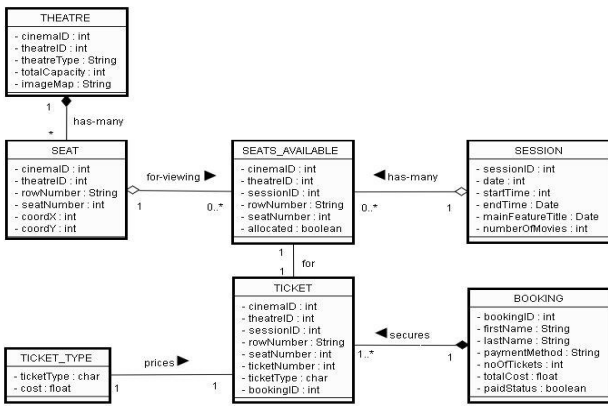
The other predicate in the body of Rule3 is *cinemaLocality* which is another *connective* predicate, the need for which is discussed above in deriving Rule1.

**3.4.4 Deriving the Ticket Assistant Rules**

Figure 6 depicts the sub-area entities involved in the *ticketAssistant* rule. The logic flow line in Figure 1 which identifies the necessary entities for the sub-area, branches in two directions, at both ends. This is partly due to the dual need to select unallocated seats in a theatre and session of choice, and book and pay for the seats that are ultimately chosen by the user. Consequently, we decided to satisfy the various sub-tasks that the Ticket Assistant agent is involved in, with two logic rules.

**Rule 4:**

The attributes/terms within this head-of-the-rule predicate are the things a customer would like to know *after* they have decided on the movie to see, and on the cinema in which to see it, such as *which seats* are available in the session. The client wanted an interface that displays a graphic 2D plan of the theatre, showing both seats that are already *allocated*, and seats that are still available. We catered for this in the design, by including an image-map of the theatre plan in the Theatre entity, and by storing the (x,y) pair of coordinates of each seat, in the Seat entity (see Figure 6). Note: seat row and number were insufficient as coordinates, since seats are usually in a curvilinear layout, often punctuated by doorways. So, in addition to the



**Fig.6 - A UML Class diagram detailing attributes for entities associated with the TicketAssistant Rules.**

various movie and cinema textual fields, the head-of-the-rule also has the name of the *image-map* file, and the *seat* (x,y) coordinates, which are relative to that image origin (0,0), which the sub-agent displays to the user for interaction. The *head-of-the-rule* is as follows:

**seatSelection**(CinemaName, Location, TheatreID, ImageMap, MovieTitle, Date, StartTime, RowNumber, SeatNumber, CoordX, CoordY, allocated) ←

Requiring the following predicates in the rule *body*, via normalized fully-attributed entities in Figure 6:

**cinema**(CinemaID, CinemaName, Location, CinemaLatitude, CinemaLongitude),

**theatre**(CinemaID, TheatreID, TheatreType, TotalCapacity, ImageMap),

**seatsAvailable**(cinemaID, theatreID, rowNumber, seatNumber, sessionID, allocated),

**seat**(CinemaID, TheatreID, RowNumber, SeatNumber, CoordX, CoordY),

**session**(SessionID, Date, StartTime, EndTime, MainFeatureTitle, NumberOfMovies),

**movieSession**(MovieID, SessionID),

**movie**(MovieID, MovieTitle, Rating, RunningTime, Website, ReleaseDate, Trailer, DvdRelease, Synopsys);

Note: The *Theatre* and *Movie* entities are shown in other figures above, so they are not repeated in Fig. 6.

This rule is used to return all seats in a session, and whether or not they are currently *allocated*. It is also capable of returning the seats' coordinates wrt the returned ImageMap file. The implementation of the Ticket Assistant sub-agent can then use these returned tuples, to display an interactive image of the theatre seating, with which the user interacts (human-in-the-loop), to select the seats they want to book.

A second rule is then used which takes the user's seat selection from the agent that encapsulates Rule4, to forward payment details. The logic rule follows:

**Rule 5:**

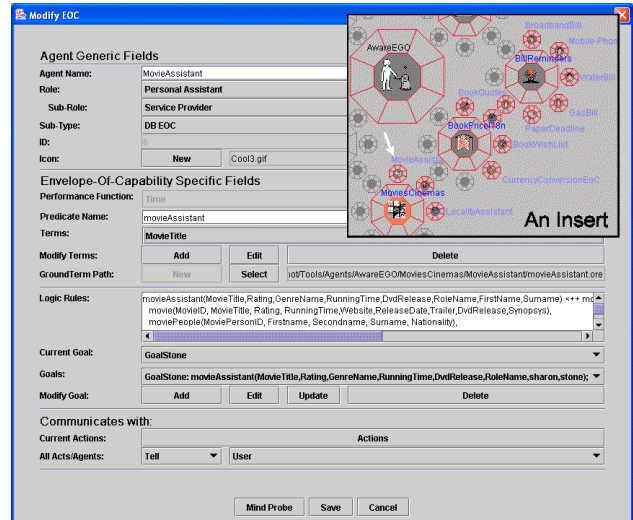
**ticketAssistant**(CinemaID, TheatreID, Date, StartTime, RowNumber, SeatNumber, TicketNumber, TicketType, Cost, BookingID, FirstName, LastName, PaymentMethod, PaidStatus) ←

It requires the following predicates in the rule *body*, via the normalized, fully-attributed entities in Figure 6:

**ticketType**(TicketType, Cost),

**ticket**(CinemaID, TheatreID, SessionID, RowNumber, SeatNumber, TicketNumber, TicketType, BookingID),

**booking**(BookingID, FirstName, LastName, PaymentMethod, NoOfTickets, TotalCost, PaidStatus);



**Fig.7 – Modification of EoC Agent properties**

The agent that wraps the ticket predicate, gets its values for *CinemaID*, *TheatreID*, *SessionID*, *RowNumber*, *SeatNumber* and *TicketNumber* from the interaction surrounding the *seatsAvailable* rule. It requires interaction with the user, to determine the *TicketType* of each ticket.

The last predicate - *booking* - needs to be encapsulated by a web service agent, that is capable of accepting payment, and returning the *BookingID* and *PaidStatus* values/terms.

#### 4. Design Implementation via the DigitalFriend

The DigitalFriend - a userfriendly MAS - is the tool we used to *implement* the design. It is especially good at incorporating a number of agent and sub-agents – including an agent type which is able to *wrap a web service* [7].

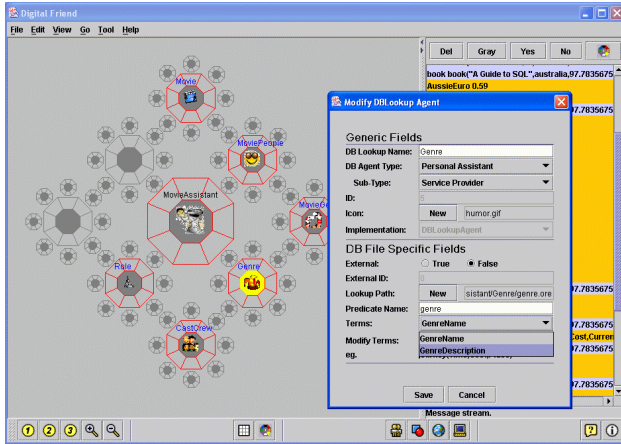


Fig. 8 - Modification of the 'Genre' sub-agent

A hierarchical view of the various agents and subagents is displayed via an octagonal-tile interface, which is recursively deep, displaying three levels of the hierarchy at any one time. The user may zoom back and forth through the hierarchy. Note: Within Figure 7 is an insert (top-right - for space reasons in this paper), which shows where the cluster of three agents are via a white-arrow, in a larger application. I.e. *MovieAssistant*, *CinemaAssistant* and *LocalityAssistant* – are grouped together under the EOC (envelope-of-capability) agent *MoviesCinemas*.

The interface – which is currently zoomed-in on the *MovieAssistant* EOC agent and its underlying sub-agents - is depicted in a screen-shot in Figure 8. Note: An EOC agent encompasses a number of sub-agents, which in turn can recursively have more sub-agents as needed [4,7]. *MovieAssistant*, and its siblings *CinemaAssistant* and *LocalityAssistant* represent the functionality available from Rules 1, 2 and 3 above in Section 3.4, and match the head-of-the-rule predicates.

You can see that the *sub-agents* of *MovieAssistant* are called *Movie*, *MoviePeople*, *MovieGenre*, *Genre*, *CastCrew* and *Role* – which exactly matches the list of predicates in the rule-body for the *movieAssistant* rule.

Both EOC agents and non-EOC sub-agents encompass predicates, which are created within the DigitalFriend interface via GUI property dialog boxes. Figure 7 represents the property dialog box for an EOC agent – specifically for *MovieAssistant*.

There are four main sub-areas of input/edit fields in the dialog: *role and identity* fields; *predicate* and *term* related fields and buttons for interaction; *logic rules* and a series of

predefined *goals/queries* for the agent; and an *inter-agent communication language* section (beyond the scope of this paper). The rule for *movieAssistant* – Rule 2, as designed in Section 3.4.2 – has been entered by the end-user into the field 'Logic Rules'. The goal/query that will return any movies involving the actress *Sharon Stone* as *star*, is the *current goal* expressed in this agent.

Each of the predicates in the rule body is represented by a sub-agent. These agents may be of various types including: agents that *wrap a web service*; *DB-lookup* (local database) agents; *java-coded* agents and *reminder-agents*. These are discussed elsewhere [4], but for space requirements and clarity sake with regard to the design method presented here, we look at one that is implemented as a *DB-Lookup agent* – namely *Genre*.

Figure 8 shows the property dialog for an agent called *Genre*. It has two *terms* – *GenreName* and *GenreDescription* – exactly matching the equivalent predicate in Rule2 derived in Section 3.4.2. It stores all movie genres – i.e. not many records/ground-terms - which is why it is implemented as a *locally stored* DB-Lookup sub-agent, rather than sourced from a web service.

Sub-agents that retrieve records of ground-terms from web-services are also represented by a predicate with a list of terms – however, they also have *timers*, set to retrieve new information at regular intervals set by the user, and they update stores of ground-terms, which they maintain locally.

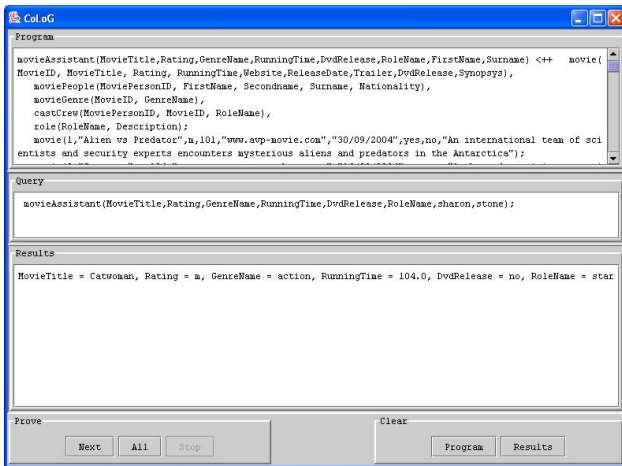
The DigitalFriend dynamically collects all rules, predicates and ground terms for an EOC agent and its sub-agents, and efficiently combines them into a logic program in the internal CoLoG language – built and fired as necessary [4]. Many of the resultant terms are passed to other agents – as specified via the inter-agent communication language in CoLoG – and many are passed as *messages* for the user's attention, as seen in the right-hand side of the background of Figure 8. The different background colors of some messages, represents a *message filtering system* within the DigitalFriend, beyond the scope of this paper. See [6].

The CoLoG program, which is dynamically constructed for a given EOC agent, can be inspected and executed (and tested and debugged) within the GUI, via the '*Mind Probe*' button at the bottom of the property dialog seen in Figure 7. The dialog that *Mind Probe* initiates, is displayed in Figure 9. In it, we can see the *current goal* has been executed via the user pressing the *Next* button, which has caused the agent to locate the movie '*Catwoman*' as one that *stars Sharon Stone*.

The only CoLoG code that the end-user had to conceive and enter into the DigitalFriend is *Rule 2*, entered via the 'Logic Rule' field in the property dialog in Figure 7. All other details have been entered via less textual GUI interaction involving buttons and single name terms - hence the need for TANDEM.

## 5. Discussion

Several MAS development environments use internal logic languages. The DigitalFriend system uses the Prolog-like CoLoG internal language interpreter to glue together various sub-agents by dynamically combining predicates, rules and ground terms. In turn, the MAS application developer needs to be able to write well-formed logic predicates and rules. We identified a gap in available methods for writing such predicates and rules.



**Fig. 9 - Mind-Probe button accesses an Agent's current state.**

ER modeling is more widely known than logic programming is, and more teachable than 'logic programming experience'. Basic task analysis is straightforward and easily learnt. ER modeling and basic Task Analysis have been combined and adapted into our presented design method for logic programs - *TANDEM* - to find appropriate predicates and their constituent terms.

As seen in the overview of the implementation of the *MovieAssistant* in Section 4, there is a significant and timely need for such design methods, in developing MAS solutions which incorporate a logic language, for modern problems that utilize web services, such as the *Movie-Cinema problem* outlined above.

Compounding the need, an MAS systems such as the DigitalFriend is aimed at the *end-user* market where people will often not be logic language programmers, but who will usually have technical skills at the spreadsheet-developer or the desktop database designer level, or the recreational programmer level. Hence the need for *strong guidance* in finding appropriate predicates as early as possible, using well-tested techniques they are either familiar with, or which can be learned in a short timeframe.

Our TANDEM design method for logic programs, complements the pattern-oriented method by Sterling [8]. Where the pattern method helps with programs that have intricate logic rules, the TANDEM method concentrates on achieving well-formed logic rules, specifically by finding

the appropriate predicates. It does this by starting with a basic Task Analysis, followed by an ER normalization of the accrued entities, then identification of *logic flow lines*, which address the primary tasks identified in the task analysis. The derivation of well-formed logic rules then becomes relatively straightforward, as demonstrated in Section 3. This results in flexible logic programs, which can be distributed amongst sub-agents in a MAS application, that can each handle specified tasks, but which also have flexible combinations of predicates and terms, to handle new unforeseen goals. The well-formed nature of the predicates and rules, means that they are in a highly adaptable form, that can be modified and added to with new rules via new sub-agents, and new web services, as new requirements unfold later on in the life of the application.

We are now in the process of developing a modeling tool, that enables a user to place logic flow lines over their ER diagram, which then automatically produces the predicates and rules, using the algorithm outlined in Section 3 above.

## 6. References

- [1] Annet, J. Hierarchical Task Analysis. In *The Handbook of Task Analysis for Human-Computer Interaction*, Diaper, D. and Stanton, N. (Eds), Lawrence Erlbaum Associates, 67-82, 2004.
- [2] Balbo, S., Ozkan, N. & Paris, C. Choosing the Right Task-modelling Notation: A Taxonomy. In *The Handbook of Task Analysis for Human-Computer Interaction*, Diaper, D. and Stanton, N. (Eds), Lawrence Erlbaum Associates, 445-465, 2004.
- [3] Benyon, D. *Information and Data Modelling, Second Edition*, McGraw-Hill, England, 1997.
- [4] Goschnick, S.B. *ShadowBoard: an Agent Architecture for enabling a sophisticated Digital Self*. Thesis, Dept. of Computer Science, University of Melbourne, Australia, 199 pages, 2001.
- [5] Goschnick, S.B., Balbo, S., Sterling, L. and Sun, C. TANDEM - a Design Method for Integrating Web Services into Multi-Agent Systems. *Proceedings, Fifth Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-06)*, pp939-41, Hakodate, Japan, May 8-12, 2006.
- [6] Goschnick, S.B. & Graham, C. Augmenting Interaction and Cognition using Agent Architectures and Technology Inspired by Psychology and Social Worlds. *Universal Access in the Information Society*, 4(3), Springer, pp 204-222, 2006.
- [7] Goschnick, S.B. and Sterling, L. Enacting and Interacting with an Agent-based Digital Self in a 24x7 Web Services World. In *Proceedings, IEEE joint conference on Web Intelligence and Intelligent Agent Technology (WI/IAT)*, Halifax, Canada, 2003.
- [8] Sterling, L. Patterns for Prolog Programming, In *Proceedings, Computational Logic: Logic Programming and Beyond 2002*: 374-401, 2002.
- [9] Sterling, L. and Shapiro, E. *The Art of Prolog*, 2<sup>nd</sup> edition, The MIT Press, Massachusetts, 1994.

