Extending Tropos for a Prolog Implementation: A Case Study Using the Food Collecting Agent Problem

Carlos Cares[1,2], Xavier Franch[1] and Enric Mayol [1]

[1] Dept. Llenguatges i Sistemes Informàtics - Universitat Politècnica de Catalunya, Jordi Girona, 1-3 08034 Ph. : 43-93 413 7839, Barcelona, Spain.
{ccares,franch,mayol}@lsi.upc.edu

[2] Dept. Ingeniería de Sistemas, Universidad de La Frontera, Av. Francisco Salazar 01145, Casilla 54-D, Ph 56-45 325000 Temuco, Chile.

Abstract. There is a recognized lack of Agent Oriented Methodologies to translate a detailed design to a software implementation; here we address this problem with a solution approach. *Tropos* is one of the most used methodologies to design agent systems and we use it to show a design for the Food Collecting Agent Problem. Our solution includes autonomous behaviour, beliefs, multiple roles playing, communication and cooperation in a simple way. We propose a method to generate a Prolog implementation from a *Tropos* detailed design, adding a step allowing relevant decisions being incorporated at design time. Besides we show how to get the Prolog implementation from this detailed design. Our experience shows that this proposal is an intuitive, direct and effective way to get a Prolog implementation for an agent system. We end the paper with illustrations about our collecting team in action.

1 Introduction

Nowadays there is a recognized lack of Agent Oriented Methodologies (AOM) at the implementation stage [1, 2]. *Tropos* [3, 4] is one of the most used AOM, however, in spite of all its virtues, its implementation stage does not have enough guidance for declarative software implementations [5]. In this paper we address this situation and propose a method to get a Prolog implementation starting from a *Tropos* design. To illustrate our method we use the Food Collecting Agent Problem (FCAP) as a case study, which is about a grid-like environment where agents can move from one slot to a neighboring slot if there is no agent already in the target slot. In this world food can appear in a randomly way in an empty slot. There is a special slot where the agents must collect the food, named the depot. In the next section we show briefly the stages of *Tropos* and our design for the FCAP, in section 3, an additional design step oriented to get a Prolog implementation is proposed and finally we show how to convert this output into a computer program.

2 Using *Tropos* for the FCAP

Tropos [3, 4] is an agent-oriented methodology for building software systems. It is adequated to describe both the social (organizational) environment and the system itself. According to [5], *Tropos* covers from early requirements to implementation with a different clear focus on each stage: (1) Early requirements focus on social context; (2) Late Requirements focus on system-to-be; (3) Architectural Design, focus on systems components; (4) Detailed Design and (5) Implementation, both focus on software agents.

Tropos uses the concepts of actors, which can be organizational, human or software; positions, roles and agents, as specializations of actors; goals and social dependencies for representing the commitments or agreements of actors (*dependees*) to other actors (*dependers*). The type of the dependency depends on the intermediary element (*dependum*) between actors. It can be *goal* (hard or soft), *plan* or *resource*. Thus the basic structure of social representation is the *dependee-dependum-depender* relationship. In the figure 1 and in the figure 2 we have illustrated the graphical representation of *Tropos* constructors according to their use. For further details about *Tropos* see [5].

In *Tropos*, at the Early Requirements stage, the analysis of the environment must be done. Since in the FCAP case study does not have a social context, we omit this stage.

At the Late Requirements stage the system-to-be is analyzed and the functional and non-functional requirements are specified. For FCAP we recognize two main actors, the food provider (FP) and the collector team (CT). There is a main dependency that represents the need of the CT from the FP for food, either initial or produced, but also the FP has constraints for the behavior of the CT in the food environment. Although there is an evolution of diagrams inside of this stage, we show the final output in the figure 1.

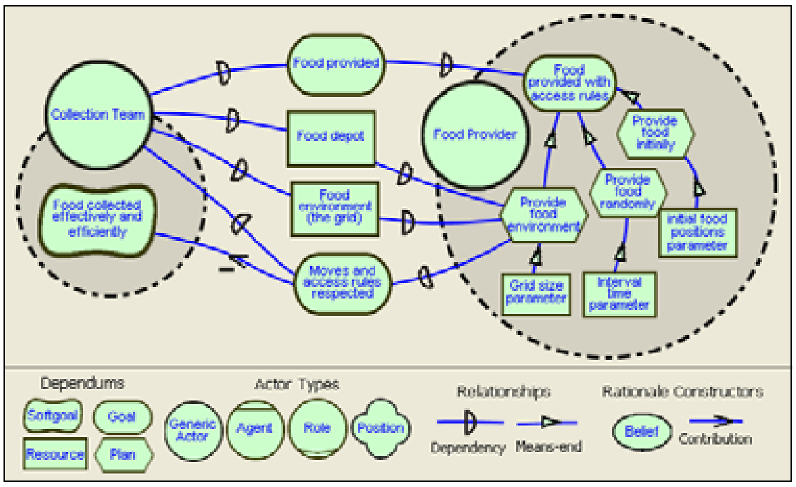


Fig.1 The output of the Late Requirements stage

In the Architectural Design stage the global architecture of the system is analyzed, new actors are incorporated and their main capabilities are identified. In our case we have added a *teamMember* position that represents all member of the team. Moreover we have decided to tackle the problem with the roles *collector* (for gathering food and disposing it in the depot) and *explorer* (for looking for food in the grid). Finally we have delegated in the *ruleGuard* role the goal to keep an adequate behavior. When we specify that the position *teamMember* covers the *ruleGuard* role, means that all members of the team must play this role. In figure 2 we show the output of the Architectural Design. For simplicity we have omitted the positive contributions from the *Team Member*, *Collector*, *Explorer* and *Rule Guard* goals to the main softgoal of the *Collection Team*.

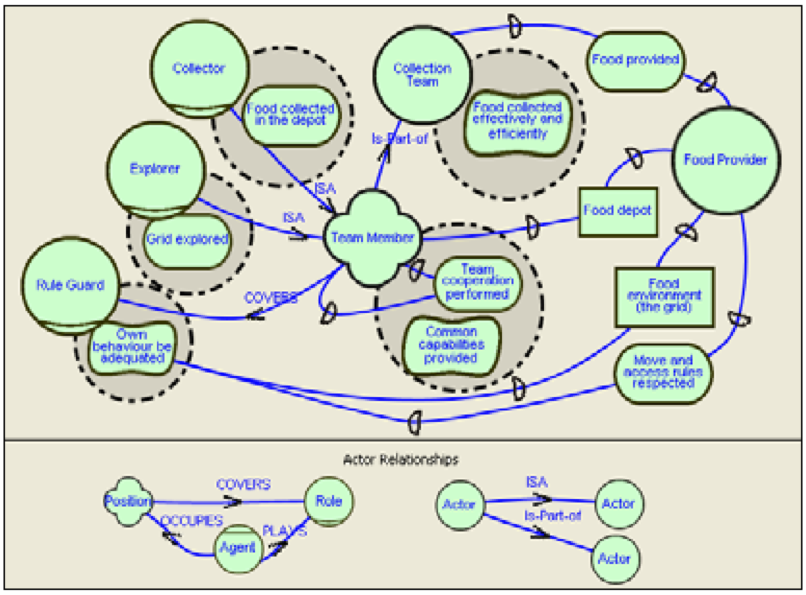

Collector
Food collected in the depot
Collection Team
Food collected effectively and efficiently
Food provided
Explorer
Is-Part-of
ISA
ISA
Grid explored
Team Member
Food Provider
Food depot
Rule Guard
COVERS
Team cooperation performed
Food environment (the grid)
Own behaviour be adequated
Common capabilities provided
Move and access rules respected
Actor Relationships
Position
COVERS
Role
OCCUPIES
PLAYS
Agent
Actor
ISA
Actor
Is-Part-of
Actor

Fig.2 The output of the Architectural Design stage

At the Detailed Design stage, each actor is individually analyzed and each goal of the Architectural Design is decomposed to specify the actor capabilities. Thus, for the *ruleGuard* role, we have designed capabilities to access the own position, to pick up food, to know empty neighbor slots and to move just into these slots. For the *teamMember* role we have identified a *belief* about the food environment, this *belief* can be updated with agents interaction, thus the cooperation among team is based on sharing their beliefs. Moreover we have provided direct access to the position of the depot and we have designed the capability to advance forward a target point in the grid. For the *collector* role we have the capabilities of disposing food in the depot, moving to the depot, moving for a guessed food position (based on its *belief*) and, in the case of no food information, looking for it in unvisited slots. We have designed a major strategic, which specify that collectors have different search spaces according the collectors quantity and the size of the food environment.

Finally for the *explorer* we have designed the capability of moving sequentially by the grid, and the necessary data resources to support this capability have been identified. In the figure 3 we show a partial view of the Detailed Design output, we illustrate this stage with the R*ule guard* and T*eam member* roles, for simplicity we have omitted the relationships and *softgoal* contributions among the actors. The simplicity of the *explorer* role, based on a sequential moving over the grid, allows focusing on less classical design aspects. In the next section we show our extension for the detailed design of the *collector* role.

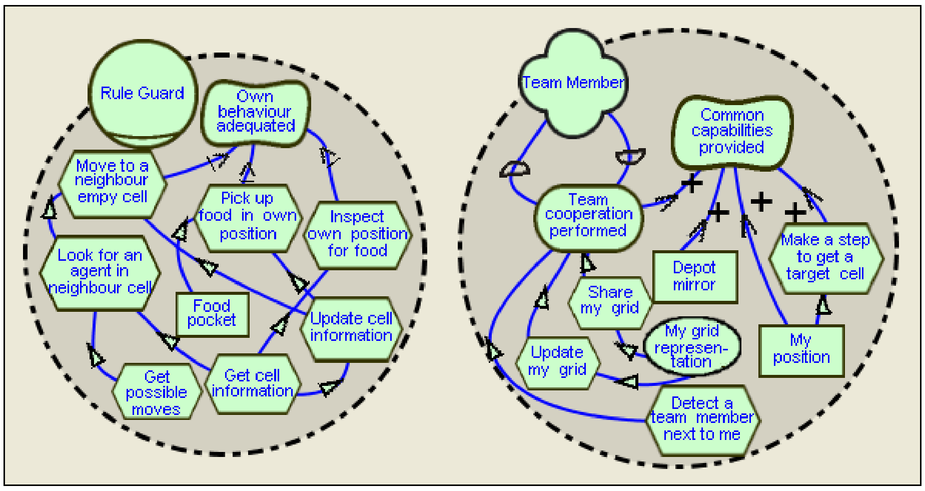

Rule Guard
Own behaviour adequated
Move to a neighbour empty cell
Pick up food in own position
Inspect own position for food
Look for an agent in neighbour cell
Food pocket
Update cell information
Get possible moves
Get cell information
Team Member
Common capabilities provided
Team cooperation performed
Make a step to get a target cell
Depot mirror
Share my grid
My grid represen-tation
My position
Update my grid
Detect a team member next to me
+
+
+
+

Fig.3 Detailed design for Rule Guard and Team Member roles

3 Extending Detailed Design for Prolog Implementation

In this section we explain how to extend the Detailed Design stage to get a declarative implementation. We propose to replace the AUML activity diagrams used in *Tropos* by scenario sequences. These have been used at the requirements stage in a web-based software system [6]. We think that it is a simple way to specify sequences when they are needed. In Prolog these sequences must have the convenient order to prove the logical goals. This has relevant logical and efficiency effects hence it is very convenient to have a representation of these decisions at design time. Besides we propose *goal* and *plan* root elements be implemented like Prolog. In the figure 4 we show a scenario sequence for the *collector* role, here the design indicates that the first goal to be proved is *put food in the depot*, but this means to check that the agent is over the depot and has food in the buffer, otherwise the second goal, *go to the depot*, should be proved, etc.

For *goal* and *plan* root elements we propose to specify the programming activation time, thus we propose four implementation attributes, namely *at begin*, *at end*, *at call* and *always*. *At call* means that the goal or plan is invocated from another goal or plan. *At begin* and *at end* indicate that the selected goal or plan should be proved just one time, at start of run time or at the end. *Always* means that the goal or plan should be permanently proved. If there is a multi thread Prolog implementation it is suggested that each goal or plan with the *always* attribute be a different thread.

Finally we propose to make the decisions about data representation. The design elements, which require having a data representation, are *resource* and *belief*. In Prolog we have two main choices: a data structure (generally a list) or the knowledge data base (KDB) included in Prolog. In our case, we have decided to represent the food environment with predicates (KDB) and the rest of *resource* and *belief* elements with lists.

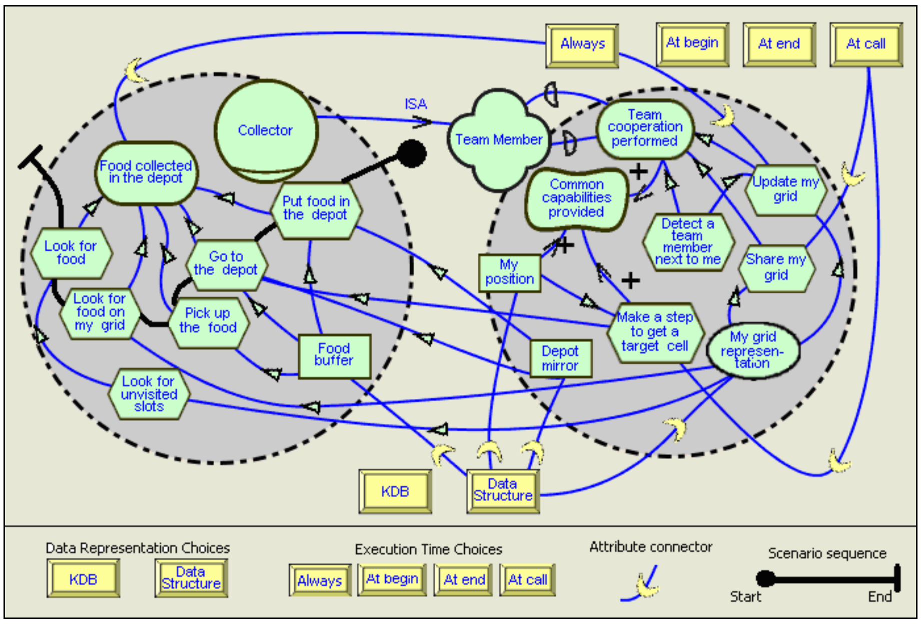
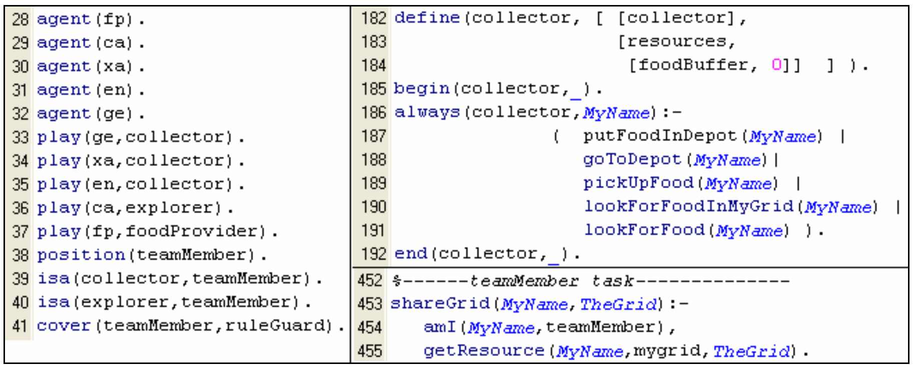


Fig.4 Partial Extended Detailed Design for FCAP

At implementation time we have a set of recommendations to translate the extended detailed design. First, make the instantiation of the problem with the predicates *agent*, *play*, *position*, *isa*, *cover*, etc., i.e. the actor types and actor relationships from *Tropos*. For FCAP we have generated a specific instantiation with five agents, a *Food Provider* agent (*fp*), an explorer agent (*ca*), and three collectors (*en*, *xa* and *ge*), we show this from lines 28 to 41 in figure 5. Second, to implement *resource* and *belief* elements as part of the *define* predicate, identifying the name of the role as first argument, and a data structure which define the resources (e.g. lines 182 to 184 in figure 5). Third, to group root elements (*goal* and *plan*) under the identified activation times, each activation time is a predicate that needs the actor type (atom) and the agent name (variable); this is illustrated from lines 185 to 192 in the figure 5. Fourth, to program the goals and a plan using a set of predicates that act over the defined data structures and clauses. This step motivates the reuse of already programmed predicates but is not an automatic step because the specific clauses depend on the semantic of the goal or plan. For example we show, in figure 5, the *amI* and *getResource* predicates that implement the *shareGrid* plan (lines 452 to 455).



```
28 agent(fp).
29 agent(ca).
30 agent(xa).
31 agent(en).
32 agent(ge).
33 play(ge,collector).
34 play(xa,collector).
35 play(en,collector).
36 play(ca,explorer).
37 play(fp,foodProvider).
38 position(teamMember).
39 isa(collector,teamMember).
40 isa(explorer,teamMember).
41 cover(teamMember,ruleGuard).
```

```
182 define(collector, [ [collector],
183                     [resources,
184                      [foodBuffer, 0]]  ] ).
185 begin(collector,_).
186 always(collector,MyName):-
187               (  putFoodInDepot(MyName)  |
188                  goToDepot(MyName)|
189                  pickUpFood(MyName)  |
190                  lookForFoodInMyGrid(MyName)  |
191                  lookForFood(MyName)  ).
192 end(collector,_).
```

```
452 %-------teamMember task--------------
453 shareGrid(MyName,TheGrid):-
454     amI(MyName,teamMember),
455     getResource(MyName,mygrid,TheGrid).
```

Fig.5. Direct implementation of the detailed design

And fifth, we have made the goals that allow the access to the set of *begin*, *end*, and *always* predicates, namely *runBegin*, *runEnd*, and *runAlways* (without arguments). It is very important to note that the definition order of agents will be the calling order, thus if we run *runBegin*, it will be executed first the *fp* begin and the last one will be the *ge* begin (according to lines 28 to 32 in figure 5). These logical goals are generic and could be used in any agent system implemented in Prolog.

The resulting system generates an output that we run under a web browser. We have set the system with an 8x8 food environment and fifteen seconds for a running. We illustrate the resulting system in figure 6 where de depot is the dark slot with the number 0 at start and 6 at the end. The food is into light slots and the agents are the slots with the strings *ca*, *xa*, *en* and *ge*.

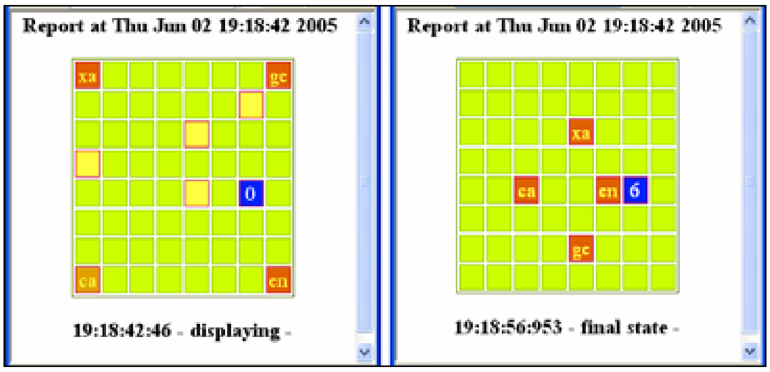

Report at Thu Jun 02 19:18:42 2005
xa
ge
0
ca
en
19:18:42:46 - displaying -
Report at Thu Jun 02 19:18:42 2005
xa
ca
en
6
ge
19:18:56:953 - final state -

Fig. 6. Running the resulting agent system

5 Conclusions

In this paper we have proposed a specific way to get a Prolog implementation from a *Tropos* design. Although we have used the FCAP, the approach is a generic developing proposal based on the atomicity of the detailed design. However a set of different projects should be carried out to get stronger evidence about its utility. Moreover this approach requires a set of generic predicates which do not have a design representation in each specific problem. However, when they have been developed, the programming is intuitive and direct. Thus a relevant part of the code could be generated automatically and the rest could be sufficiently documented for programming aid.

About FCAP we have designed a solution using multiple roles playing (*Team member*, *Rule guard* plus *Collector* or *Explorer*). The agents cooperate in the solutions sharing information about their belief of the world. Besides, the common behavior is grouped in the *Team member* role, being an efficient solution to the problem. Our experience indicates that this proposal is an intuitive, effective and efficient way to implement agent-oriented systems.

Acknowledgement

This work has been done in the framework of the research project UPIC, ref. TIN2004-07461-C02-01, supported by the Spanish Ministerio de Ciencia y Tecnología. Carlos Cares wants to acknowledgment to MECE-SUP FRO0105 Project of the chilean government and the University of La Frontera, who supports his stand at Universitat Politècnica de Catalunya. Besides we want to thank to Francesca Toni and Paolo Torroni for the conceded grant to expose our report at CLIMA 2005.

References

1. Dastani M., Hulstijn J., Dignum F., Meyer J.:Issues in Multiagent Systems Development. Third International Conference AAMAS04, Columbia, USA, July (2004), 920-927.
2. Hoa, K. and Winikoff, M.: Comparing Agent-Oriented Methodologies, In the proceedings of the Fifth International Bi-Conference Workshop on Agent-Oriented Information Systems, Melbourne, (at AAMAS03), July (2003).
3. Giorgini P.,Perini J., Mylopoulos J., Giunchiglia F, Bresciani P.:Agent-oriented software development: A case study. In Proceedings of the Thirteenth International Conference on Software Engineering – Knowledge Engineering (SEKE01), Buenos Aires (2001).
4. Perini A.,Brisciani P.,Giunchiglia P., Giorgini P., Mylopoulos J.: A knowledge Level Software Engineering Methodology for Agent Oriented Programming. In Proceedings of the Fifth International Conference on Autonomous Agents, Montreal, Canada, May (2001).
5. Sannicoló, F., Perini, A., Giunchiglia, F.: The Tropos modelling language. A User Guide, Technical report DIT-02-0061, University of Trento, February (2002).
6. Liu L. and Yu E.:Designing Web-Based Systems in Social Context: A Goal and Scenario Based Approach, Lecture Notes in Computer Science Vol 2348, Springer-Verlag, Berlin Heidelberg New York (2002), 37-51.