

Collecting Gold

JIAC IV Agents in MULTI-AGENT PROGRAMMING CONTEST

Axel Hessler, Benjamin Hirsch, and Jan Keiser
 {axel.hessler|benjamin.hirsch|jan.keiser}@dai-labor.de

DAI-Labor, Technische Universität Berlin, Germany

1 Introduction

This contribution (description and implementation) has been prepared by members of the *Competence Center Agent Core Technologies* of DAI-Labor at Technische Universität Berlin. We use the JIAC IV agent framework with accompanying toolkit, which have been created in the course of several projects at DAI Labor, intended for telecommunications and telematics services to be implemented quickly and effectively, and to be administered reliably.

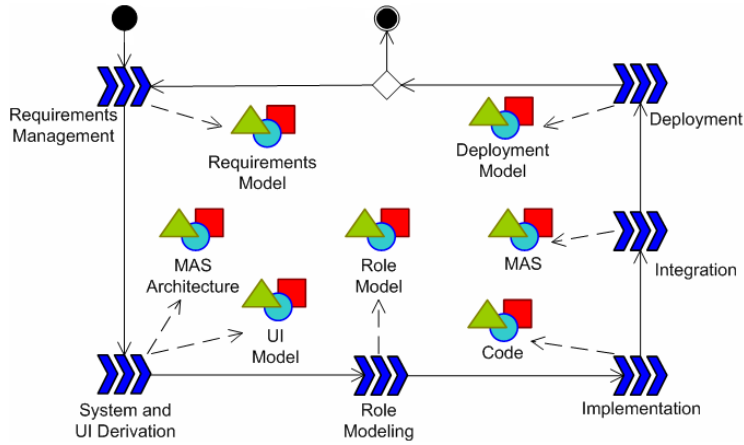


Fig. 1. JIAC methodology - iterative and incremental process model in SPEM [4] notation

2 System Analysis and Design

2.1 JIAC IV Methodology and Tools

The JIAC IV agent framework comes with its own customised methodology and a number of tools integrated in the Eclipse IDE.

As shown in Figure 1, the development process starts with collecting domain vocabulary and requirements, which then are structured and prioritised. Second, we take the requirements with the highest priority and derive a MAS architecture by listing the agents and create a user interface prototype. The MAS architecture then is detailed by creating a role model, showing the design concerning functionalities and interactions. We then implement plans, services and protocols, which are plugged into agents during integration. Agents are deployed to (one or more) agent platforms and the application is ready to be evaluated. Depending on the evaluation we align and amend requirements and start the cycle again with eliminating bugs and enhancing and adding features until we reach the desired quality of the agent-based application.

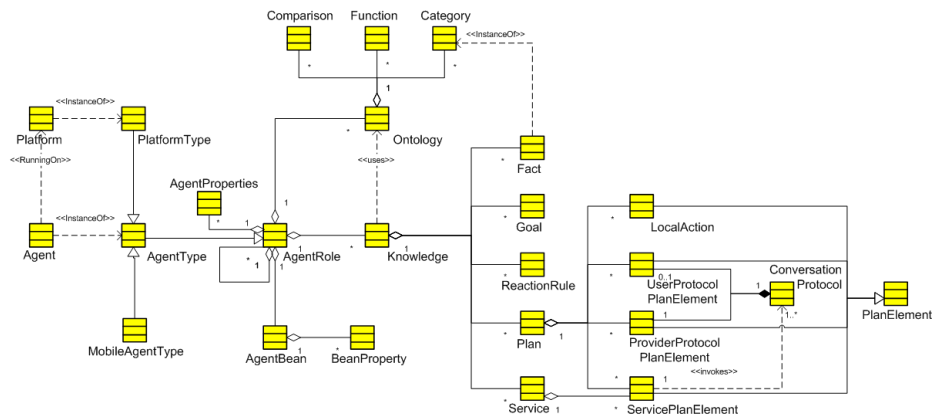


Fig. 2. JIAC IV MAS meta-model

The JIAC methodology is based on the JIAC MAS meta-model (Figure 2). JIAC has explicit notion of goal, rule, plan, service and protocol. Knowledge written in JADL and AgentBeans written in Java constitute agentroles, which are plugged into standard JIAC agents. The standard JIAC agent is already capable of finding other JIAC agents and their services, using infrastructure services and provides a number of security and management features.

For any part of the JIAC meta-model we provide an editor (source code as well as visual editor) for easy agent and application development. Reuse is supported by a plugin that allows search and retrieval of components and solutions. A context sensitive help and a number of interactive tutorials complete the JIAC IV toolbox.

2.2 Solution

The design follows our methodology with a number of iterations. We use our tools which always provide a textual and modelling perspective of all agent concepts.

In the following we sketch the application of the methodology to the problem of implementing the competition agents. We distinguish the design of single agent behaviour and cooperation.

Single Agent Behaviour We start collecting the simulation domain vocabulary and create the ontology containing such concepts as nuggets, gold-digger, grid cells, and so on. Listing 1.1 shows the *GoldDigger* category with its attributes. Furthermore, basic features such as the ability to communicate with the simulation server and a simple path-finding algorithm are created.

```
(cat GoldDigger (ext TemporalGridObject)
  (name string)
  (currentPosY int (init -1))
  (currentPosX int (init -1))
  (teammate bool)
  (carriesGold int (init 0)))
(intention Intention)
```

Listing 1.1. Extract from GoldWorld ontology

In a further iteration some higher level plans are designed, embodied into special roles such as *Explorer* or *Transporter*. In particular, we have behaviours for finding gold, moving to a certain position, picking up gold, and scoring. The explorer role should have capabilities to systematically search the terrain for gold, the transporter role brings the gold to the depot. Listing 1.2 shows a first version of the *makePoint* plan of the transporter role.

```
(act makePoint (var ?score:int ?self:GoldDigger ?x:int ?y:int ?world:GridWorld)
  (pre (and
    (att score SIMULATION (fun Int_DAI_1.sub ?score 1))
    (att self SIMULATION ?self)
    (att currentPosX ?self ?x)
    (att currentPosY ?self ?y)
    (att world SIMULATION ?world)
    (att hasGold (fun getCellFromGridWorld ?world ?x ?y) true)))
  (eff (att score SIMULATION ?score))
  (script (var ?depot:GridCell ?depotX:int ?depotY:int)
    (seq
      // pick gold
      (goal (and (att self SIMULATION ?self) (att carriesGold ?self true)))
      // goto depot
      (eval (and (att depot SIMULATION ?depot) (att posX ?depot ?depotX) (att posY ?depot ?
        depotY)))
      (goal (and (att self SIMULATION ?self) (att currentPosX ?self ?depotX) (att
        currentPosY ?self ?depotY)))
      // drop gold
      (goal (and (att self SIMULATION ?self) (att carriesGold ?self false)))
    ) ) )
```

Listing 1.2. Higher-level plan “makePoint”

Furthermore, we exchange our path finding capability with a generic A* implementation as our local search path finding algorithm does not work in labyrinths. In Figure 3 the principle control flow of a single agent is shown, which should work well with the simulation environment.

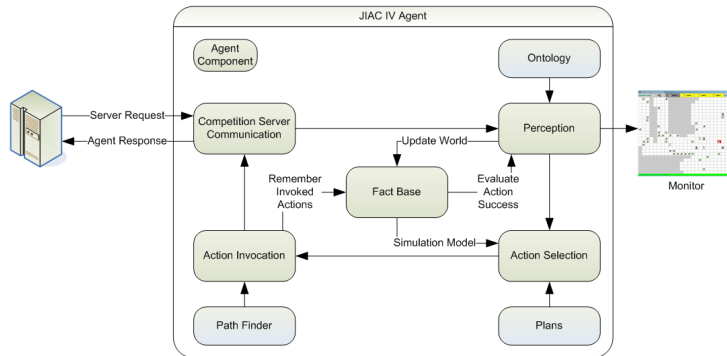


Fig. 3. Customising the JIAC IV standard agent for the contest

Multi-agent Cooperation When dealing with MAS we always assume that the MAS must be worth more than the sum of its parts. We address this with a number of iterations dealing with communication, coordination and cooperation.

Agents cooperate on a number of levels. First, they share their perception. Next, we enable our agents to share their agent state (e.g. that the agent carries no gold items) and intentions (such as “I plan to pick gold at X,Y”) as they may choose to go to the same unknown field or to pick the same gold item. Now every agent can appraise from what it knows if it will be better to leave the team member alone or to take the intention as its own when its more promising.

Our approach to communication and cooperation is fully decentralised. Each agent has the capability for finding the other agent on the network. It then directly tells every agent about its perception, agent state and intentions. There is neither a message broker nor a central instance, which coordinates the contest agents. Every agent builds its own world model from what it is told by the server and the other agents. Every agent also plans for itself, taking the states and intentions of its teammates into account.

Concerning scalability, autonomy and distribution it does not matter whether JIAC agents run on the same computer or not at application level. Without writing a single line of code you may design different distribution patterns, from an agent platform on one computer to a totally distributed system where every agent has its own processor. Or the agent even may chose at runtime to migrate to another computer, if there is less load, to have more computing power for solving the A* in a very large environment.

3 Software Architecture

3.1 JIAC IV Agent Framework

Java Intelligent Agent Componentware (JIAC) is based upon the CASA BDI architecture described in [1]. It combines a scalable component framework, a

knowledge representation toolkit, a control architecture and an agent infrastructure. Additional features are a runtime environment, system agents, tools, and libraries [2].

JIAC provides the JADL (JIAC Agent Description Language) programming language. Based on three-valued logic, it incorporates ontologies, FIPA-based speech acts, a (procedural) scripting part for (complex) actions, and allows to define protocols and service based communication. Rather than only relying on a library of plans, the framework also allows agents to plan from first principles [3].

4 Conclusion

We have shown that it is very easy and effective to solve the simulation problem using the JIAC IV agent framework. We first collect the domain vocabulary and basic requirements from the scenario description, derive a monitor GUI and basic agent architecture. Then we gather basic capabilities and interactions in roles and implement them. Implemented roles are plugged into the agents and deployed on the agent platform. After evaluating the performance of our agents we collect new requirements, bugs and feature enhancements and start a new iteration.

4.1 Open Issues

There are still some issues left. Our agents behave fair, at the moment, in that they give way for other agents even if they are opponents. We also assume that the current approach to coordination does not scale very well as it takes $n * (n - 1)$ connections with n the number of agents in the team. Observation of opponents would be fine as well as to guess what they plan and then to crisscross it. There is evidence that this can be solved with some more iterations following the methodology.

References

1. Sessler, R.: Eine modulare Architektur für dienstbasierte Interaktionen zwischen Agenten. PhD thesis, Technische Universität Berlin (2002)
2. Fricke, S., Bsufka, K., Keiser, J., Schmidt, T., Sessler, R., Albayrak, S.: A Toolkit for the Realization of Agent-based Telematic Services and Telecommunication Applications. *Communications of the ACM* **44**(4) (2001) 43–48
3. Konnerth, T., Hirsch, B., Albayrak, S.: JADL — an agent description language for smart agents. In Baldoni, M., Endriss, U., eds.: *Declarative Agent Languages and Technologies IV*. Volume 4327 of LNCS., Springer Verlag (2006) 141–155
4. : Software Process Engineering Metamodel (SPEM) Specification. Version 1.1. Object Management Group, Inc. (2005)