Herding Cows with JIAC V

Axel Heßler, Benjamin Hirsch, and Tobias Küster
{axel.hessler|benjamin.hirsch|tobias.kuester}@dai-labor.de

DAI Labor / Technische Universität Berlin
Ernst-Reuter-Platz 7
D-10587 Berlin
Germany

Abstract. In this article we report on the JIAC V team for the agent programming competition 2009, going through the different phases of development and describing the JIAC V agent framework. Based on an iterative approach, we identified and implemented different agent roles. While there is no explicit team concept, the agents cooperate by informing each other of their perceptions and intentions, which leads to emergent team behavior which very dynamically and flexibly reacts to the state of the game.

1 Introduction

One of the central activities of the Competence Center *Agent Core Technologies* of *DAI-Labor* at Technische Universität Berlin is the development of the JIAC agent framework family (Java Intelligent Agent Componentware), merging intelligent agents with aspects known from Service Oriented Architecture (SOA). Besides our current agent framework JIAC V [1], we are also developing MicroJIAC [2], which is tailored towards resource-limited devices, such as mobile phones and embedded components. Last but not least, the now discontinued JIAC IV [3] is still in use in many projects, especially in the field of telecommunication and telematics.

We welcome the annual agent programming contests as an opportunity for testing and improving the strength and usability of our JIAC agent frameworks and the accompanying methodologies and tools.

– In 2007, we deployed two teams to the contest: JIAC IV and MicroJIAC. For MicroJIAC, this was one of the first 'field tests' [4], and in the end, MicroJIAC made it for the second place – outscored only by JIAC IV [5].
– The next year we used the contest to introduce JIAC V to the community – which was still called by its internal name JIAC TNG back then [6]. While the scenario was significantly more complex, JIAC TNG's debut performance was a great success, placing first again.
– This year, we participated with two teams again, JIAC V and MicroJIAC [7]. To make things more interesting, and to see how people unfamiliar with our frameworks can use them, the teams have been prepared by students of a

university course at TU Berlin [8]. From this we got some fresh ideas, and, with a little help by members of the development team, made the third win in a row with JIAC V.

All in all, from each contest we have gained some valuable insight in our agent frameworks, tools, and their application.

2 System Analysis and Specification

When developing multi-agent systems, we follow the iterative and incremental methodology MIAC specific to the JIAC agent framework. The methodology has been developed in the context of industrial and teaching projects and reflects more then 10 years of experience in agent-oriented software engineering. It has undergone several iterations itself until we came up with a practical guide to agent development, which covers the necessary aspects of the development lifecycle, and which is flexible enough to be adopted to organisational structures and cultures. The methodology can be extended and supplemented by sophisticated methods from software engineering, modeling or quality assurance.

The following sections give a short introduction to MIAC. A detailed description of the JIAC V multi-agent system will follow in Section 4.

2.1 The MIAC Methodology

In MIAC, the development process starts with collecting domain vocabulary and requirements, which then are structured and prioritized. Second, we take the requirements 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 the several agent beans, which are plugged into the agents during integration. Agents are deployed to agent nodes 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 (see Figure 1).

2.2 Role Modeling

As mentioned above, this year our contribution to the contest has been implemented by students of a university course. All students took intuitively a role-based approach to analysis and design, a role meaning the aggregation of functionality and interactions regarding a certain aspect of the domain. The following roles have been identified. (Note that we name and explain the roles regardless of whether they have been implemented or not in the end.)

First of all, the agents need to explore their environment in order to get to find cows, find all obstacles in order to calculate the best way to their own corral, and find the opponent's corral. This is what is subsumed in the *Explorer*

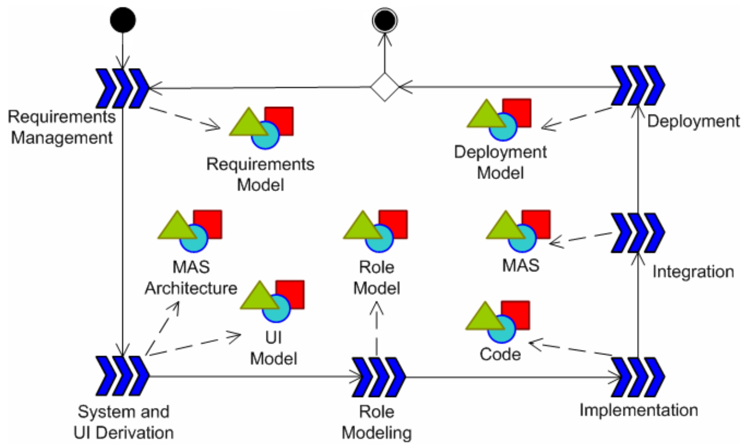

Fig. 1. JIAC methodology – iterative and incremental process model in SPEM [9] notation

role. We then need to drive one or more cows to the corral, the *Herder* role. We also assume that cows may escape from the corral when someone is using the fence switch, so we presumably need a *Keeper* role. The additional feature of this year's scenario, the switch, leads to another role: the *ButtonPusher*, although we expect that this is not a full time job.

We also identified implicit roles which all agents must be capable of: connect to the server, receive perceptions from and send actions to the server: the *Server-Connector* role. An agent must also be capable of parsing the server message and update its world model, the *Perception* role. The perception role also notices if actions failed or not. Furthermore, each agent should be capable of talking to all other agents of its own team to share its perception and its intention, the *TeamCommunicator* role.

A third group of roles that has been identified concerns the opponent: analyze opponent behavior in the *OpponentAnalyzer* role. Based on the analysis the agents can then interfere with opponent agents' actions, and the *TroubleMaker* role is born. If applicable to the situation, the own agents may try to steal cows from the other corral. These capabilities and interactions can be concentrated in the *Thief* role. We also must take into account that the opponent has the same skill, so we will have to expand the Keeper role with the ability to prevent that opponent agents steal our cows. When discussing the roles, there was no consensus on the Thief and TroubleMaker roles. Some stated that it is not worth to have these extra roles, because when making trouble and stealing these agents could have driven cows to one's own corral.

Last but not least, we identified the necessity to analyze the behavior and performance of our own team, the *TeamAnalyzer* role.

For modeling the several roles, the Agent World Editor (AWE) has been used [10] (see Figure 2). These roles were then aggregated into agents and set on the agent node. The role model was used for generating the agents' XML configuration files. Each role corresponds to one or more agent beans and may include other roles, each one providing one aspect of the roles behavior by providing respective actions, or implementing observers, listening for changes to the agent's memory.

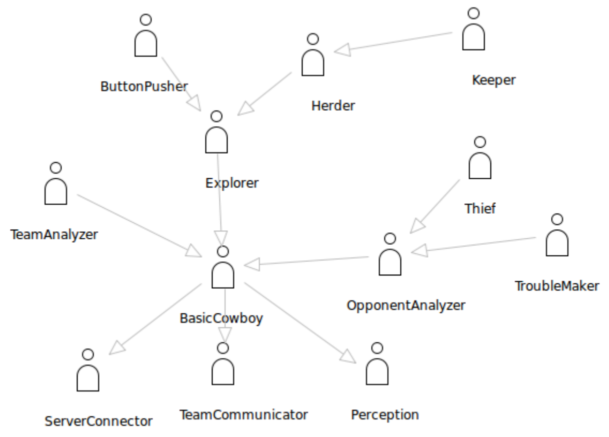

Fig. 2. Role Model

3 System Design and Architecture

Given the roles and functionalities that were identified during the analysis phase, the design of the agents consisted of a number of components that allowed to fulfill the requirements. Recall however that since JIAC follows an agile and incremental approach, requirements are prioritized and can be changed even late in development to gain a competitive advantage[1].

The JIAC methodology creates graphical representations of the agents and associated roles and components, as well as an agent configuration which is in principle executable by the framework. This rapid prototyping supports the incremental design approach taken by the JIAC methodology with early and continuous delivery of a valuable MAS.

The first iteration defined an execution cycle and a number of components used by the agents, namely:

– server connection

[1] See also: "Principles behind the Agile Manifesto". http://agilemanifesto.org/principles.html

– message parser
– world model
– decision component

While the first two components are necessary to connect to the contest server, they are not really interesting in the context of this article. Instead we will focus on the world model and the decision component.

3.1 The World Model

The world model reflects the agent's (incomplete, possibly outdated) view on the world. It is instantiated from the ontology, which defines the concepts and their relationships. Within the execution cycle, a number of actions are taken. First, messages from the environment are parsed, then, the agent's world model is updated, and finally, a decision on the next move is made.

Figure 3 shows the initial world model. In it, the main actors of the environment are reflected, divided into movable and unmovable objects. Agents are distinguished as belonging to the own or opponent team, as are the corrals. It turned out however that the model as initially designed was too heavy on some details and the performance of the systems was also not good.

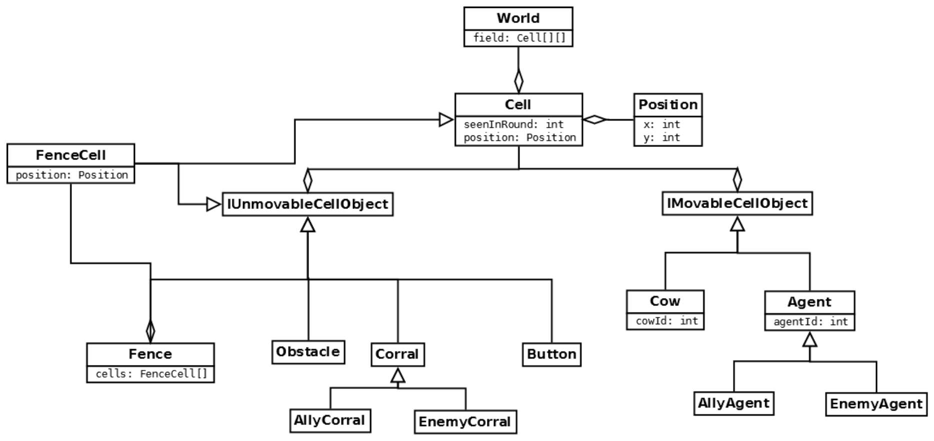

Fig. 3. Initial world model

Therefore, in another iteration, the world model was simplified and all objects were made to be directly accessible from the main world object. Basic intentions were added to the model, too. Figure 4 shows the resulting world model (without attributes for readability).

A *World* object holds information about the grid *cells*, *cows*, *fences*, *corrals* and *agents*, the computed *actualPath* and the *ownIntention* of the agent holding

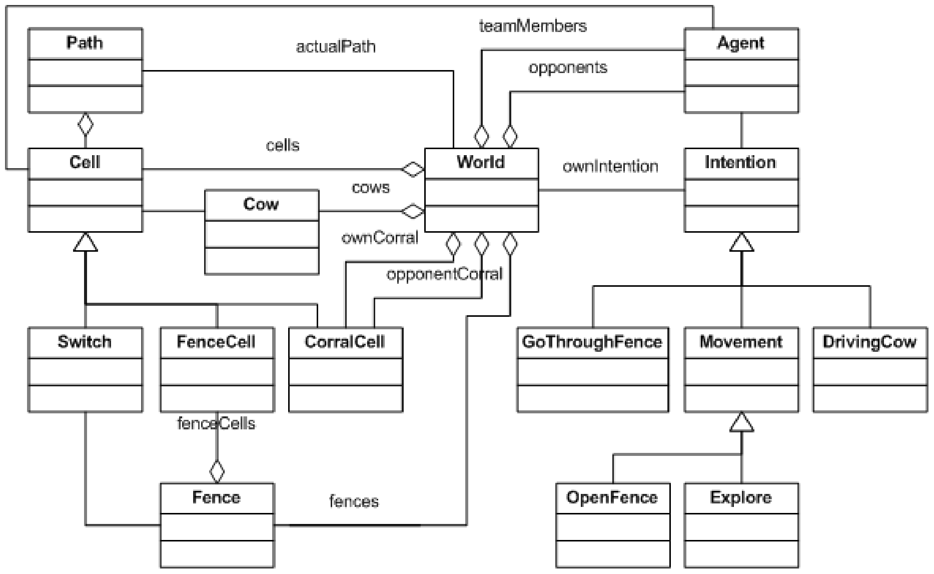

Fig. 4. Final world model

this world object. There are special cells, the *Switch* and the *FenceCell*, which together constitute a *Fence*. *CorralCell*s form *corrals*.

Perceptions are not part of the world model. They are designed straightforward from what is given in a server message and directly update the agent's world model. They are also sent to the team members in order to update their world model, too.

*Intention*s in the world model store additional, domain specific information about what the agent wants to do next and are the outcome of the decision component. An intention here is not just "I want to move west". The specialisation of intention in the world model roughly reflect the roles that are taken by the agent:

- **Explore:** contains the movement information and the number of unknown cells, which the agent expects to uncover there
- **DrivingCow:** reveals the exact cow the agent wants to drive
- **GoThroughFence:** just says that the agent wants to go through the fence given in this intention
- **OpenFence:** the agent wants to open the fence given in this intention

An intention updates the agent's own world model and is also sent to the team members, so that they can take into account the intention of the sender in their decisions. The *ownIntention* is then converted into a server action (e.g. "north"). In a next simulation step, the agent can check whether the action was successful or not, and he will decide on whether to stick to the current intention or reject it.

3.2 The Decision Component

Again, following the incremental and iterative approach, the initial decision component is just a random move. This allows us to make sure the execution cycle runs reliably. Once the “outer shell” of the agent is implemented, we then add incrementally more complex behavior.

In the first extension, a behavior for finding the next unknown cell is implemented. This is then again extended by adding an A* algorithm to find the shortest path to this cell. If cows are found, we calculate a path from the closest, to the agent, cow to the corral and drive the cow by positioning the agent behind the cow, i.e. the opposite direction of the cow path.

Once this basic behavior is functional, team communication is added to the system. Each agent broadcasts its perceptions as well as its intentions to the others. The decision component now not only takes into account the private perceptions, but decisions are based on a “global” team world model. Each decision bean now calculates for the single agent on which agent should drive identified cows, based on the distance of agents to cows and cows to the corral. Since the intentions of other team members are known, it is made sure that no agents try to drive the same cow.

Finally, roles are implemented. Following the priorities, we have first implemented the roles *Explorer*, *ButtonPusher* and *Herder*. These behaviors are selected by simple reaction rules based on the current state of the world model, as shown in the following pseudo code.

```
IF no free cow known
   THEN adopt Explorer role
IF free cow known and not herded by other agent
   THEN adopt Herder role
IF cow and herding agent close to fence and self closest to fence
   THEN adopt ButtonPusher role
```

4 Programming Language and Execution Platform

We implemented the team using the JIAC V agent framework, which we will describe in this section.

JIAC V is aimed at easy and efficient development of large scale and high performance multi–agent systems. It provides a scalable single-agent model and is built on state-of-the-art standard technologies. The main focus rests on usability, meaning that a developer can use it easily and that the development process is supported by tools.

The framework also incorporates concepts of service oriented architectures such as an explicit notion of service as well as the integration of service interpreters in agents. Interpreters can provide different degrees of intelligence and autonomy, allowing technologies like semantic service matching or service composition. JIAC V supports the user with a built-in administration and management interface, which allows deployment and configuration of agents at runtime.

An important aspect of JIAC is that is offers different levels of programming. For example it is possible to implement agents using only Java. On the other and, agents can also be programmed using only the scripting language JADL++, which allows to easily define services and incorporates OWL [11] as data description language [1]. As described in the preceding section we have opted to use only Java to implement the contest agents.

The JIAC V methodology is based on the JIAC V meta-model and derived from the JIAC IV methodology. JIAC V has explicit notions of rules, actions and services. Composed services are modeled in BPMN [12] and transformed to JADL++. We distinguish between composed services and infrastructure services. The former can be considered a service orchestration with some enhancements (e.g. service matching) whereas the latter describes special services that agents, composed services, or basic actions can use, e.g. user management, communication or directory services. Rules may trigger actions, services or updates of fact base entries. Basic actions are exposed by AgentBeans and constitute agent roles, which are plugged into standard JIAC V agents. The standard JIAC V agent is already capable of finding other JIAC V agents and their services, using infrastructure services, and it provides a number of security and management features.

The core of a JIAC V agent consists of an execution cycle that is responsible for executing services (see Figure 5).

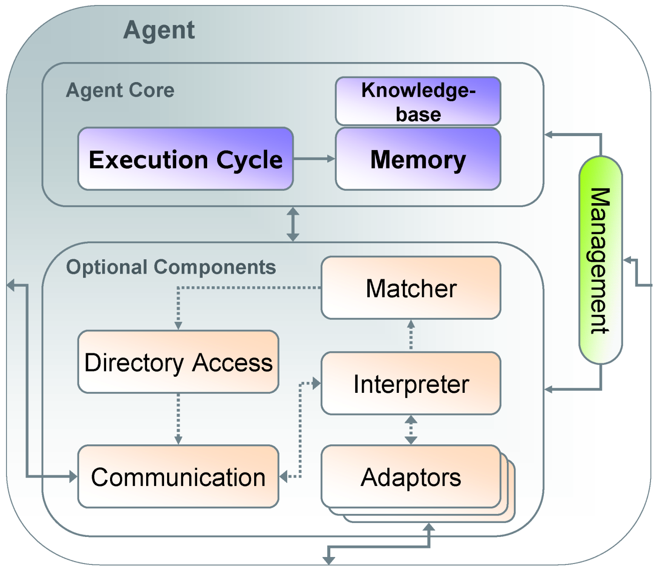

Fig. 5. The architecture of a single agent

Our approach is based on a common architecture for single agents in which the agent uses an adaptor concept[2] to interact with the outside world. There exists a local memory for each agent to achieve statefulness, and each agent has dedicated components (or component groups) that are responsible for decision making and execution.

In JIAC, the adaptor concept is used not only for data transmission, but also for accessing different service technologies that are available today [13]. Thus, any call to a service that is not provided by the agent itself can be pictured as a call to an appropriate effector. Furthermore, the agents' interpreter allows to execute a set of different services. These services' bodies may also contain calls to different services or subprograms. Consequently, an agent is an execution engine for service compositions.

In the following, we will give you a brief explanation of the function of each component:

- **Matcher:** The Matcher is responsible to match the invoke commands against the list of known services, and thus find a list of applicable services for a given invoke. The service templates within the invoke commands may differ in completeness, i.e. a template may contain a specific name of a service together with the appropriate provider, while others may just contain a condition or the set of parameters.
  Once the matcher has identified the list of applicable services, it is up to the interpreter to select a service that is executed. Note that this selection process includes trial&error strategies in the case of failing services.
- **Memory:** The interpreter uses the agent's memory to manage the calls to services as well as the parameters. We use a simple *Linda*-like tuple space [14] for coordination between the components of an agent. Additionally, the current state of the execution can be watched in the memory any time by simply reading the complete contents of the tuple space, allowing for simple solutions for monitoring and debugging.
- **KnowledgeBase:** The knowledge base provides functionalities for reasoning and inferences within an agent. All declarative expressions within either a service description or an action invocation are evaluated against this knowledge base. In contrast to the Memory above, the knowledge base is a semantic memory rather than a simple object store and has a consistent world model.
- **Interpreter:** The interpreter is the core for service execution. It has to be able to interpret and execute services that are written in JADL++. Essentially, all atomic actions that can be used within the language are connected to services from either the interpreter or the effectors of the agent.
- **Adaptor:** The adaptors are the agent's connection to the outside world. This is a sensor/effector concept in which all actions that an agent can execute on an environment (via the appropriate effector) are explicitly represented by an action declaration and optionally a service declaration that is accessible for the matcher. Thus, all actions may have service descriptions that are equivalent to those used for actual services.

[2] Each of these adaptors may be either a sensor, an effector, or both.

5 Agent Team Strategy

Due to the similarity of the scenarios, large parts of the strategy could be adopted, which we had developed for the contest in 2008 [6]. To clear this up-front, we did not use any sophisticated, explicit team concept, neither shared plans, commitments and plan alignments, nor peer-modeling techniques. Our approach is more similar to a decentralized team AI as described in [15]:

> "...interactions between squad members, rather than a single squad leader, determine the squad behavior. And from this interaction, squad maneuvers emerge. This approach is attractive because it is a simple extension of individual AI..."

Firstly, every agent builds its own world model from what it is told by the server (i.e. its own perceptions) and its team mates. Therefore, each agent broadcasts its perceptions and intentions (see 3.1) to the rest of the team. Every agent also plans for itself, without a central point of control, by taking the intentions of its team mates into account. Thus, by sharing both their perceptions and their intentions, redundant actions could be prevented to a large extent. Further, it allows for quickly re-entering the simulation in case an agent should need to be restarted.

Just like 2008, the agents navigated using the A^* algorithm, which was used for both, calculating its own path and for calculating the path a cow should take and where to position itself to drive the cow in this direction. Thus, this algorithm provides basic capabilities for navigation, obstacle avoiding, cow herding, and exploration. The algorithm is slightly adopted using higher costs for neighbouring cells of obstacles (you cannot drive a cow away from an obstacle). Movable objects in the scenario, such as cows and agents, are seen as obstacles as long as the algorithm terminates with a path. This change results in the agent behavior of collision avoiding. Agents start exploring the field, until they find a cow, which they will then try to drive to the corral. The nearest agent takes the job of the herder and sends its intention to its team mates. The other agents keep exploring the field until they find their own cow or everything is discovered.

This year, due to the improved cow algorithm on the server side, single cows are more difficult to drive by a single agent, while it is easier now to drive smaller flocks and at least possible to drive even huge, compact herds. Just like last year, we could observe some emergent behavior here, as the agents will pick one of the cows closest to themselves and to the corral, which, since cows seldom stand alone, automatically leads to a group of agents driving the flock of cows closest to the corral, which works better than a single agent driving a cow or a flock.

While we planned to have one agent to exclusively watch that no cows escape the corral, a trial simulation showed that this behavior will be covered emergently, too: Whenever a cow escaped the corral, the agent closest to the corral will drive it back inside, while the other agents can go on with their assignments. The basic cow driving behavior made this agent the 'Keeper'.

Also, there was much debate among the JIAC-team developers about whether to implement a 'thief' role, or not. In the end, we decided not to implement a thief, but not to prohibit such behavior, either. Thus, when it turns out that the cows in the enemy corral appear to be those that are easiest to drive to the own corral, our agent would not hesitate 'stealing' them – in fact, they were fully ignorant of them being in the enemy corral in the first place – but they will not do so to damage the enemy. Since we had an issue with 2008's opponents blocking our agents, we decided not to implement such kind of behavior, and are happy that our agents did not emergently develop such behavior, either.

One feature of the scenario made changes in the cowboy agents' behavior necessary: fences. We solved the problem by introducing two new intentions: *OpenFence* and *GoThroughFence*. With these two intentions our agents tell their team mates if they just open a fence for others to drive cows through or if they just want to go through the fence in order to explore the world and find new cow herds. Thus, whenever an agents wants to go through a fence, for exploration or when driving cows, it will broadcast its intention to go through the fence, and the closest allied agent will open the fence for him. This led not only to teams of herders, autonomously driving flocks of cows through fences, but also to teams of two explorer agents, helping each other passing through the fences while exploring the map.

For future editions of the contest, we plan to extend our team's coordination capabilities to further reduce redundant actions and to optimize paths walked by the agents. Further, we think about analyzing the opponent, e.g. which cows the enemy agents are likely to engage next.

6 Technical Details

Our development infrastructure, both for the contest and for the development of the JIAC family of agent frameworks in general, includes version control with Subversion, the Apache Maven build system, a Continuous Integration Server, bugtracking software and a Wiki for capturing ideas.

All of our agents ran on a single multi-core Windows machine which was dedicated for this job during the week of the contest. During the games, the agents were very stable, and due to the constant synchronization of the several agents' knowledge and intentions a crash could quickly be compensated, as the crashing agent's knowledge would not be lost, allowing him to quickly re-enter the running game. However, we experienced some problems *between* the games, when switching from one map to another. Here, our agents repeatedly crashed and had to be restarted on the new map. Apart from this, our agent framework was very stable.

7 Discussion and Conclusion

In general, we have at least three very positive results concerning the contest:

– The contest is an excellent testbed for debugging and benchmarking our JIAC agent frameworks.

– The contest is an excellent platform for innovation and promotes sustainable development of multi-agent systems, frameworks and tools
– The contest is an excellent scenario for teaching AO principles.

Let us amplify this: In preparation for the contest we have found and fixed many bugs in the lifecycle and execution cycle of our agents, and in the updating and interpretation of the world model. Due to the high requirements of the contest we tuned several core components concerning performance and reliability, making code easier and working more efficiently, which is a benefit also for maintenance and other projects that use JIAC (e.g. the $NeSSi^2$ project [16], a JIAC-based simulator, measured the overall performance of their application: improved performance by factor 8!). The contest was the first real application for JIAC TNG/V in 2008. And, finally, in preparation for the contest we implemented many features that make the life of the JIAC programmer easier (easier to learn, easier to use, easier to debug, easier to deploy).

Although we have won the contest, we still think there is room for major improvement in single algorithms, single agent behavior, and, of course, in team behavior. The greatest pleasure was, again, to see emergent team behavior while agents are driving cows, keep cows in the corrals and "steeling" cows from the other team. We have reproduced this behavior in this year's contest and we have now a clue how emergent behavior can be "engineered". But for all that, we would like to compare emergent behavior to an explicit team concept with explicit roles, dynamic role assignments and changing groups following a common, shared goal.

7.1 Short guide to be competitive

At the very end, we want to give some hints to newcomers. We here sketch what we did every time in our three contest participations, and it seems to be successful. This is our personal point of view and may not be generalized. It is more a rough procedure than a scientific methodology. We think we also can generalize with caution that this procedure is independent from concrete programming language, agent framework or development methodology, although choosing familiar ones may help during implementation.

– In every contest, we took part in, a single agent consisted at least of five components: *Server Connection component, Message Parser component, World model, Decision component, Team Communication component.* So we assume that this is a valid modularization. These components also helped us to distribute the implementation work over more programmers if there were any.
– There seems to be a suitable control cycle that is similar to the stateful agent model (model-based reflex agent [17]): *Receive perception, Parse, Update world model, Decide, Send action.* In our case, the cycle is triggered by the server message, except for the initial server registration process.

- We always start development with implementing this control cycle with dummies of parser, world model, and decision component, sending a random move as action.
- We then continue with implementing the parser and ontology, which makes up the world model.
- When the cycle runs reliably, we add real behavior: find next unknown cell. We then add A* to find the shortest path to next unknown cell. There are several implementations of A* available in any language.
- If our agents now find a cow, they calculate the path from the cow to the corral. We place the nearest agent behind the cow. If it is in corral the agent will find the next cow and so on.
- If one does not have a centralized approach to agent team (meaning that the agents share *one* world model) the agents should share their perception (what they got from the server) and intention (what they want to do). This will improve the basis of decision-making for every single agent and will also boost the team performance tremendously.

We see the above bullet point as necessary steps to be competitive, and beginners following this procedure will reach a position already in middle field of the contest competitors.

- When we come to this point, we start adding details and fine-tuning: *Open fences, Go through fences, What is the best unknown cell (to explore the world efficiently), Learning the cows' behavior, Special tactics*. We collect the ideas, prioritize them and then follow this list while improving our agent team implementation.
- Finally, we make sure that our world model is cleaned up when one simulation ends and the other one starts in order to survive more then one simulation in a row.

References

1. Hirsch, B., Konnerth, T., Heßler, A.: Merging Agents and Services — the JIAC Agent Platform. In Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A., eds.: Multi-Agent Programming: Languages, Tools and Applications. Springer (2009) 159–185
2. Patzlaff, M., Tuguldur, E.O.: MicroJIAC 2.0 - The Agent Framework for Constrained Devices and Beyond. Technical Report TUB-DAI 07/09-01, DAI-Labor, Technische Universität Berlin (July 2009) http://www.dai-labor.de/fileadmin/files/publications/microjiac_20_2009_07_02.pdf.
3. Fricke, S., Bsufka, K., Keiser, J., Schmidt, T., Sesseler, R., Albayrak, S.: A Toolkit for the Realization of Agent-based Telematic Services and Telecommunication Applications. Communications of the ACM **44**(4) (April 2001) 43–48
4. Tuguldur, E.O., Patzlaff, M.: Collecting Gold: MicroJIAC Agents in MULTI-AGENT PROGRAMMING CONTEST. In Dastani, M., Segrouchni, A.E.F., Ricci, A., Winikoff, M., eds.: ProMAS 2007 Post-Proceedings. Volume 4908 of LNAI., Springer Berlin / Heidelberg (2008) 257–261

5. Hessler, A., Hirsch, B., Keiser, J.: JIAC IV in Multi-Agent Programming Contest 2007. In Dastani, M., Segrouchni, A.E.F., Ricci, A., Winikoff, M., eds.: ProMAS 2007 Post-Proceedings. Volume 4908 of LNAI., Springer Berlin / Heidelberg (2008) 262–266
6. Hessler, A., Keiser, J., Küster, T., Patzlaff, M., Thiele, A., Tuguldur, E.O.: Herding agents - jiac tng in multi-agent programming contest 2008. In Hindriks, K.V., Pokahr, A., Sardina, S., eds.: Programming Multi-Agent Systems. 6th International Workshop, ProMAS 2008, Estoril, Portugal, May 13, 2008. Revised Invited and Selected Papers. Volume 5442 of Lecture Notes in Artificial Intelligence., Springer (2009) 228–232
7. Hessler, A., Küster, T., Niemann, O., Sljivar, A., Matallaoui, A.: Cows and Fences: JIAC V - AC09 Team Description. In Dix, J., Fisher, M., Novák, P., eds.: Proceedings of the 10th International Workshop on Computational Logic in Multi-Agent Systems 2009. Volume IfI-09-08 of IfI Technical Report Series., Clausthal University of Technology (2009)
8. Sljivar, A., Niemann, O.: Service Engineering mit Agenten. Projektausarbeitung Team TNG, SS 2009. Technical report, DAI-Labor, TU Berlin, Germany (2009)
9. Object Management Group: Software Process Engineering Metamodel (SPEM) Specification. Version 1.1. Object Management Group, Inc. (January 2005)
10. Lützenberger, M., Küster, T., Heßler, A., Hirsch, B.: Unifying JIAC agent development with AWE. In: Proceedings of the Seventh German Conference on Multiagent System Technologies, Hamburg, Germany, Springer (2009)
11. McGuinness, D.L., van Harmelen, F.: OWL Web Ontology Language. W3C Recommendation (2004) http://www.w3.org/TR/owl-features/.
12. Object Management Group: Business Process Modeling Notation, V1.1. Final Adopted Specification formal/2008-01-17, OMG (January 2008) http://www.omg.org/spec/BPMN/1.1/PDF.
13. Hirsch, B., Konnerth, T., Hessler, A., Albayrak, S.: A serviceware framework for designing ambient services. In Maña, A., Lotz, V., eds.: Developing Ambient Intelligence (AmID'06), Springer Paris (2006) 124–136
14. Gelernter, D.: Generative communication in linda. ACM Transactions on Programming Languages and Systems **7**(1) (1985) 80–112
15. van der Sterren, W.: 5.3 Squad Tactics:Team AI and Emergent Maneuvers. In: AI Game Programming Wisdom. Charles River Media (2002) 233–246
16. DAI-Labor, TU Berlin, Germany: $NeSSi^2$: Network Security Simulator http://www.nessi2.de/.
17. Russel, S., Norvig, P.: Artificial Intelligence: A Modern Approach. 2nd Edition edn. Prentice Hall (2003)

A Summary

In order to compare the systems properly, we will have a short compilation of the answers to the initial questions of each system **at the end of the special issue**. So this part of the appendix will not be part of your paper.

Please include in this appendix short answers (1-3 lines at most) to the given questions. You can **in addition** refer to the precise part of your paper, where you have more space for discussion.

1.1 At DAI Labor/TU Berlin, we are developing the JIAC family of agent frameworks, specifically JIAC V and MicroJIAC, merging agents with SOA.

1.2 We joined this contest for testing and improving JIAC and its accompanying methodology and tools.

1.3 The JIAC agents ran on a single four-core machine running Windows XP and Java.

2.1 We started by defining the ontology and continued with a role-oriented analysis and by identifying respective roles.

2.2 For role modeling our AWE tool has been used. Our role model also includes basic roles such as *Perception*, *Server Connector* and *Team Communicator*.

2.3 We used the JIAC specific methodology *MIAC*, which is illustrated in Section 2 of our article.

2.4 Roles are used for design and configuration. While communicating their perceptions and intentions to other agents, they autonomously decide which role to take at which time. There was no central coordination; like last year, team work developed emergently.

2.5 JIAC is a true multi-agent systems; in the contest distributed coordination mechanisms were used.

3.1 Using AWE, agents are composed of roles, which themselves consist of components, each of which reflects an AgentBean. For interaction, agents use JIAC's message multi-cast feature, and for the server communication respective adapter components are used (effector and passive sensor).

3.2 The MIAC methodology was used for design, too. MIAC is a pragmatic, fast and efficient MAS development and is used throughout the entire agent development cycle.

3.3 In JIAC, each agent has its own thread of control and decision capabilities. Further, JIAC agents can communicate with each others, which is used for exchanging their perceptions and intentions. This way, Team work forms emergently, without central control.

4.1 JIAC is based on the Java programming language. For development, we use the Eclipse IDE, enriched with JIAC specific plugins and tools. We also use version control, continuous integration, bugtracker and wiki.

4.2 The JIAC framework has explicit representations for actions (services and capabilities), messages and knowledge, agents, agent roles, agent nodes, components (beans), and many more.

4.3 Using AWE, role models are exported to Spring configurations, where agents, nodes, roles and components each are represented as Spring Beans. Ontologies are converted to Java classes; however, we are currently working on integrating OWL directly.

4.4 Packages: `beans` – effectors implementing roles; `connection`, `parser` – server environment; `ontology` – representation of world model; `path` – path finding.

4.5 Each agent tries to drive the nearest cow, that is not already driven by another team member, to the corral, or otherwise explores the field until he finds one. Agents help each other through fences.

5.1 The A^* algorithm is used for navigation, obstacle avoiding and herding cows (calculating the cow's path and positioning the agent accordingly). Blocking and robbing have not been implemented.

5.2 There was no team coordination strategy, except for communicating perceptions and intentions and helping each others through fences. Agents just tried to find and drive the nearest idle cow; team work developed emergently.

5.3 If two agents intend to drive the same cow, or an agent has to open a fence, the closest one is given precedence (utility optimization).

5.4 The agents constantly exchanged their perception (their view of the world) and their intentions (what they planned to do next).

5.5 Most information was multi-casted to the whole agent team, yielding $n * (n-1)$ individual communications of perceptions and intentions each turn.

5.6 While the emergent behavior worked out this year, it might not be sufficient if additional complexity is introduced. Thus, we are planning to have a more explicit team concept next year, maybe with a coordinator agent. Also, we think about anticipating and actively interfering enemy agents' plans.

6.1 No background processing has been done.

6.2 In case of a crash, communication between the agent and the server is reinitiated. The loss of knowledge of that agent is largely compensated by multi-casting the other agents' perceptions.

6.3 During the several matches, our system was very stable; however, we experienced some crashes between games, when switching to the next map.

7.1 We take a pragmatic approach to agent engineering, focusing on the problem to solve rather than how to employ as many agent concepts as possible. Building upon last year's observations of emergent team behavior, we developed a simple yet successful multi-agent system.

7.2 We once more experienced how very helpful it is to use agent modeling and an agent framework when analyzing, designing, and implementing the system. Further, it was interesting to see how last year's emergent behavior could be reproduced in this year's contest, despite e.g. the changes in the cow algorithm.

7.3 Having our *own* agent framework, we had no difficulties in choosing which framework to use. Still we've had many discussions on the approach, e.g. what and when to communicate among the agents, prioritizing strategies, etc.

7.4 As we've seen, even this year's scenario could be handled easily with simple rules of behavior and even without an explicit team strategy. For next year's contest we wish to see even more complexity and more possible actions, requiring e.g. long-term planning or more complex communication.