CORE DESIGN PATTERN FOR EFFICIENT MULTI-AGENT ARCHITECTURE

Kasper Hallenborg

Abstract: Interaction engineering is fundamental for agent based systems. In this paper we will present a design pattern for the core of a multi-agent platform - the message communication and behavior activation mechanisms - using language features of C#. An agent platform is developed based on the pattern structure, which is legitimated through experiences of using JADE in real applications. Results of the communication model are compared against the popular JADE platform.

Keywords: multi-agents, design pattern, C# language features, message based architecture, behaviors.

ACM Classification Keywords: D.2.11 Software Architectures, D.2.13 Reusable Software.

Conference: The paper is selected from International Conference "Intelligent Information and Engineering Systems" INFOS 2008, Varna, Bulgaria, June-July 2008

Introduction

Multi-agent systems, which also are referred to as *Distributed Artificial Intelligence* (DAI) [Weiss, 1999], are naturally expected to act and collaborate in a distributed environment and across different hosts. Thus from the beginning agent platforms have focused on supporting communication between agents across networks.

FIPA has proposed an abstract architecture for agent organization, and the dominating part focuses on message transport and agent communication. These specifications concern different levels of the communication, from the low-level message transport and communication protocols to the higher level abstract speech act theories for the message content. An increasing number of agent platforms try to comply with these specifications, and we find some of the most popular agent platforms, like JADE and FIPA-OS, in this category, given that they aim to support all kind of multi-agent systems.

Particularly in control systems, but for many other applications as well, the agents are just virtual representations of real entities in the application environment. The agents may have a simple bound communication channel to their physical entity, or the commands of the control unit are sent to PLCs or robots in the production environment, which performs the real actions. Thus in many situations, there is no need for advanced network communication support, as the agents are running in the same execution environment, often also in shared memory. Making just a few assumptions we can boost the performance of messages handling between agents by eliminating the overhead off network communication. Many existing multi-agent platforms try to avoid this overhead by grouping agents running on the same machine, but the abstract and general message envelopes still impact the scalability of the platform, when the communication increase.

In this paper we will present a design pattern and implementation details for a backbone to a multi-agent platform. The presented code listings for the implemented pattern is coded in C# taking advantage of language features that are not directly supported in Java. Most open-source platforms are Java based, but a few .Net based platforms are now available, such as MAPNET, EtherYatri.Net, Agent-Service and CAPNET. The latter two also being FIPA compliant, but Java is still being the dominating language for the open-source community.

In the next section we will elaborate on how JADE, as one of the most popular agent platforms, handles communication and messages.

Related Work and Motivation

Advanced interaction mechanisms are what really distinguish multi-agent platforms from general distributed architectures. Thus, a key feature of multi-agent systems has always been the mechanisms that allow the agents able to make intelligent decisions about their interactions and can participate in interactions that not necessarily were foreseen at design time [Jennings, 2000].

Many agent platforms implement this high level of abstraction by using an agent communication language (ACL) based on the speech act theory, originally introduced by Searle [Searle, 1969]. Thus most agent platforms are based on some kind of message-based interaction, where the message format allows abstract content to be encoded. FIPA-ACL specified by FIPA seems to be the most commonly used in new agent platforms [FIPA, 2002].

Aiming for more cognitive and deliberating agents more advanced interaction frameworks have been build on top of the message languages, such as goal-directed interactions [Cheong, 2006][Braubach, 2007] inspired by the BDI architecture, or role-based interactions [Cabri, 2006], but has not matured yet. Many agent platforms are mostly concerned about the basic message handling, message format, and encoding/decoding.

The JADE agent platform from Telecom Italia Lab [Bellifemine, 2005] is among the most popular agents platform available today, and it is rather generic in how it handles interactions, and we had extensive experience with it, from implementing control a baggage handling system in an airport in Asia [Hallenborg, 2006]. Thus, JADE is an appropriate candidate for explaining the message handling principle of agent frameworks. Our experience with JADE for real life applications was also that for many application domains, the rather abstract handling of message semantics was not required at all, even though you still want the agent mechanisms. This is specially the case for manufacturing systems and other agent-based systems, which are less spontaneous and more or less all participants in interactions are known at design time, at least from their characteristics and capabilities.

Similar to other agent platforms JADE has en rather simple and intuitive communication model based on asynchronous message passing. In JADE each agent lives inside a container, and a JADE platform refers to such a set of distributed containers, though several containers are allow on the same machine.

As illustrated in figure 1 each agent has a sort of mailbox – a message queue, where received messages are dropped and the agent is notified, but the receiver is in full control of if; when, and how to react to a new message. The common approach of using JADE specifies that each agent has a number of attached behaviors, which are activated if the message template matches an incoming message. After executing the action of the behavior, the behavior object will either be detached or start waiting for new incoming messages, depending on its configuration.

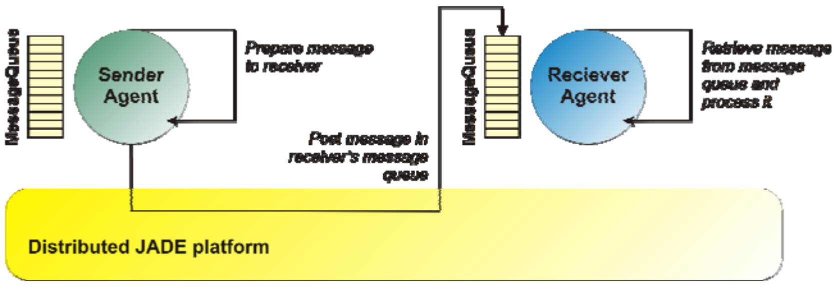
MessageQueue
Sender Agent
Prepare message to receiver
Post message in receiver's message queue
MessageQueue
Reciever Agent
Retrieve message from message queue and process it
Distributed JADE platform

Figure 1: Communication model for the JADE platform

Our experience of applying JADE to this large real life application left no complaints about the architecture and communication model of JADE. The serious performance problems were due to the internal handling of messages inside JADE during the message passing process. Running a profiler on the implemented system everything pointed at the *send* method of the Agent class as the performance leak. Digging into the source of JADE the problem could be revealed from an extensive set of tasks being executed every time a message is sent.

In the short version the *send* method of *Agent* delegates the call to the *handleSend* method of *AgentContainer-Impl* class that always requires a clone of the message[1], not only for a message with multiple receivers, but for a single receiver as well. Besides making a deep clone of the message object, which include several byte array copies of the many string fields in a message, the message is wrapped in a generic command object that is processed after credentials of the sender is set in the message. The processing step includes a number of filtering steps of the message, which perform some extensive checking of the message before it is actually sent to the receiver. The extensive copying and processing is just one part of the problem, and usually irrelevant if the communication is low, but it scales very badly when the communication level increase. At the bottom this is due to numerous of synchronized methods in the filtering and processing steps, which slows the execution caused by massive scheduling.

All this is done even if both sender and receiver is within the same container in the same executive space, where message handling should be as simple as moving a memory reference, under just a few assumptions. And this performance flaw has nothing to do with the semantics or abstraction level of the content being encoded into the message, the content could be a simple as an integer and the procedure would still be the same.

For the real life applications we are working with, such as the baggage handling problem, the performance overhead in the communication model kill all advanced decision logic of the agents, and even with some modifications to the JADE source implementation, it has become evident that a much simpler approach should be pursued.

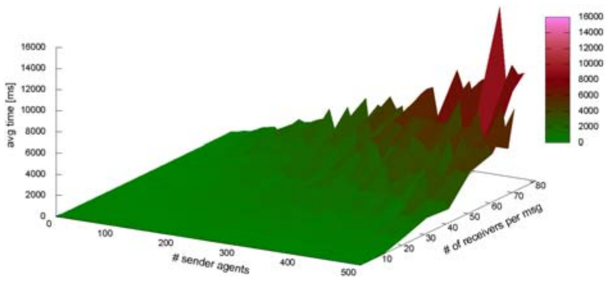

Figure 2: Average message sending time – many agents

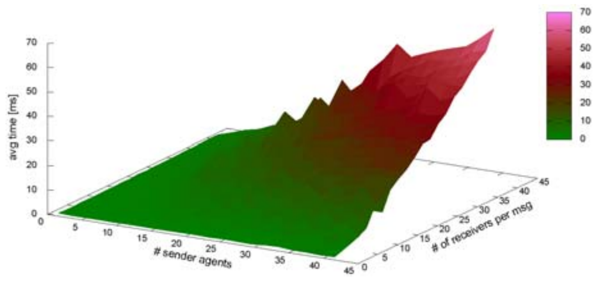

Figure 3: Average message sending time - few agents

Figure 2 and 3 show how JADE scales with the communication level in a very basic test application with the purpose of just sending and receiving simple messages. In large systems (figure 1) with up to 500 agents the average message sending time is kept under 22 ms if all 500 agents sends 1 message concurrently to a random receiver from a pool of 500 receivers, but it increases dramatically if we increase the number of receivers for the same message. If the 500 agents send the message to 80 receivers the average message time the time is approximately 10 seconds on average for just a single message. The measured time reflect solely the time an agent spends on executing the *send* method.

Behavior based Agent Architecture

We were quiet satisfied with the simple behavior and communication model of JADE, so similar features should be available in a new approach as well. Instead of doing serious hacking in the JADE source, we decided to create a simple agent platform from scratch, which were implemented under two important assumptions, at least for the first version.

- All agents were running in the same execution space with shared memory
- Receivers would not modify received message objects

[1] One of the parameters to *handleSend*. The *send* method of Agent is final, so it cannot be overridden.

With all agents in shared memory on the same machine, remoting could be avoided and message transfers could be handled by moving a simple memory reference. Especially due to the second assumption, because there would be no need to clone message objects if receivers do not modify the content.

The choice of using C# over Java was based both on ease of system integration with the rest of our application setup, but mainly due to the advantageous language features about events and delegations. With only a few modifications the pattern can also be implemented in Java.

The architecture is basically the same, as illustrated in figure 4. Agents are still connected to a container, but at the moment, there is no support for distributed containers. The container manages all the agents and put messages into the message queue of a receiver, and the same message object goes into all receivers.

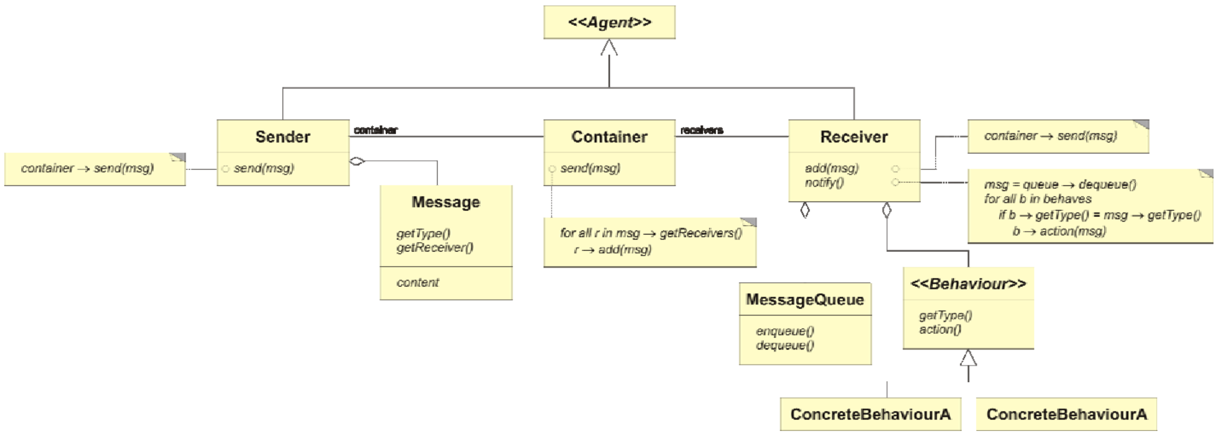

Figure 4: Pattern Structure

Message passing is in focus in figure 4, where both sender and receivers are instances of an agent class. Sender passes a message object to the container, which look up all the receivers and add the message object to their message queue. All receivers are then awaked by a notification and messages from the message queue will be processed, which means that for all behaviors attached to the current receiver the action method of the behavior will be invoked if the message type matches the type of the message.

Thus the communication principle is rather straight forward and is based on a pure asynchronous communication model. The sequence diagram of figure 5 illustrate the message being sent through the container and end up in the message queue of the receiver, which notifies and wakes up the receiver agent. The receiver agent then extracts the message from the queue and activates the appropriate behaviors in its own thread, and the sender agent can continues its own task as soon as it has delivered the message.

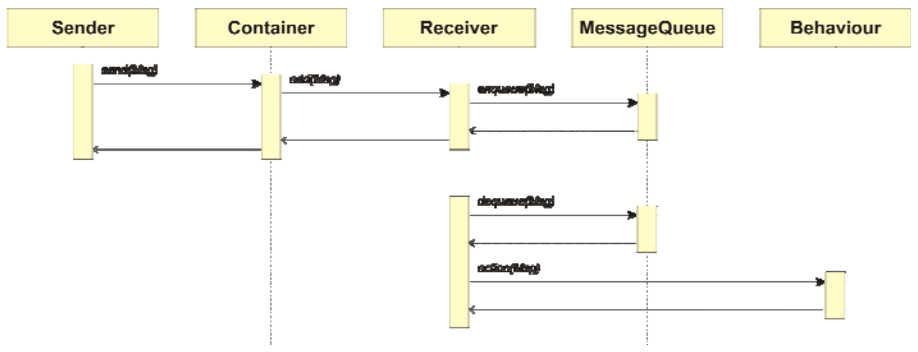

Figure 5: Sequence diagram for the communication model

Implementation

The pattern design presented in figure 4 represents the core architecture of the agent platform, which basically provides a communication platform for the agent-based systems. Therefore we also constrain the discussion of implementation details to this part of the agent platform.

In order to make the platform valuable the communication must be lightweight and the event mechanisms for both notifying receivers and activating behaviors must be very efficient.

Agent body: Sending message from the sender through the container management is rather straight forward. Instead it is not trivial to decide how receivers should activate their behaviors based on incoming messages.

All agents have their own thread and if not active performing some tasks, they will be sleeping, waiting for input, such as message. The main run loop of the thread body of a basic agent is given by the listing below

```
 1:        private void MainLoop()
 2:        {
 3:            m_Alive = true;
 4:            while (m_Alive)
 5:            {
 6:                m_SyncEvent.WaitOne();
 7:                while (m_MessageQueue.Count > 0)
 8:                {
 9:                    Message msg = null;
10:                    lock (m_MessageQueue)
11:                        msg = m_MessageQueue.Dequeue();
12:
13:                    if(m_Behaviours.ContainsKey(msg.GetType()))
14:                    {
15:                        BehaviourActionMsgDelegate behaviours = m_Behaviours[msg.GetType()];
16:                        if (IsBehavioursAsync)
17:                        {
18:                            foreach (BehaviourActionMsgDelegate bh in behaviours.GetInvocationList())
19:                                bh.BeginInvoke(msg, null, null);
20:                        }
21:                        else
22:                            behaviours.Invoke(msg);
23:                    }
24:                }
25:            }
26:        }
```

The activeness of the agent is controlled by a WaitHandle (*m_SyncEvent* in line 6), which can be signaled whenever new messages in the message queue are ready to be processed. When the agent is awakened it will process all messages in the message queue before it falls asleep again.

For each message the agent checks if it has behaviors that match the received message type (line 13). *m_Behavior* is a mapping of *BehaviourActionMsgDelegate* objects, which extends the *MultiCastDelegate* C# type that can chain delegates. So whenever a new behavior is added to an agent, it is just added the particular chain of the message types it matches.

This chain of delegates will be invoked with the current message object either sequentially or by asynchronous calls (line 16-22) based on the *IsBehaviourAsync* property of the Agent that can be modified by the programmer.

Behaviors: The consequence of invoking behaviors, using the simplified principle of delegates as presented above, is that behaviors must have a single invocation point, which can be activated after instantiation. JADE has a similar approach with the *action* method that runs the behavior, so the core task can be isolated from instantiation, recycling, etc. of the behavior object.

Delegates, which are method pointers in C#, are a perfect mechanism to achieve an even more flexible solution, which is not tied to a specific method name. For convenience programmers are still encouraged to implement invocation points as methods with the name *action*. Thus a simple behavior could look like

```
public class ConcreteBehaviour : Behaviour
{
    ConcreteBehaviour(...) { ... }  // Constructor

    public void Action(Message msg)
    {
        // TODO : the tasks of the behavior
    }
}
```

Remember that behaviors are matched against the message type, so an efficient way for a behavior to match a message is to provide action methods that take subtypes of *Message* as their single argument. This also overcomes another classic problem with JADE of having behaviors that should react on different messages. In JADE the problem can be solved by a more complex message template, or synchronizing behaviors with a shared data store to transfer data between the behaviors, which is rather non-intuitive for beginners.

With this approach the programmer simply add action methods to the behavior object for each of the message types to be matched, as exemplified below

```
public class ConcreteBehaviour : Behaviour
{
    ConcreteBehaviour(...) { ... }  // Constructor

    public void Action(ConcreteMessageA msg)
    {
        // TODO : the tasks of the behavior when message type A is received
    }

    public void Action(ConcreteMessageB msg)
    {
        // TODO : the tasks of the behavior when message type B is received
    }
}
```

The final thing missing is how delegates for the invocation points of the behavior are coupled to the agent. In order to make it as simple as possible for the programmer, an *addBehaviour* method on an *Agent* provides a standard way of adding a behavior to the agent.

```
 1:        internal protected void AddBehaviour(Behaviour behave)
 2:        {
 3:            IEnumerator<Type> enumerator = behave.GetEnumerator();
 4:            while (enumerator.MoveNext())
 5:            {
 6:                if (!m_Behaviours.ContainsKey(enumerator.Current))
 7:                    m_Behaviours[enumerator.Current] = behave[enumerator.Current];
 8:                else
 9:                    m_Behaviours[enumerator.Current] += behave[enumerator.Current];
10:            }
11:            behave.MyAgent = this;
12:        }
```

The behavior super class simply supports iteration by implementing the *IEnumerable<Type>* interface, so we can loop through all the message types that a behavior will respond to, and they are added to the mapping of behaviors in the agent (line 7 and 9).

The only part missing to be explained is how the delegates are created and how we can enumerate the invocation points for any sub types of the behavior class. As shown in line 7 and 9 above we use the special *indexer* language construct of C# to get a delegate of a certain type for a behavior. The indexer create the delegate for the right *action* method

```
public BehaviourActionMsgDelegate this[Type msgType]
{
    get
    {
        foreach (MethodInfo mthInfo in this.GetType().GetMethods())
        {
            if (mthInfo.Name == ACTION_METHOD_NAME)
                if(mthInfo.GetParameters()[0].ParameterType.Equals(msgType))
                    return delegate(Message msg) { mthInfo.Invoke(this, new object[] { msg }); };
        }
        return null;
    }
}
```

We simply use reflection to find the right method that takes the correct message parameter, create a new delegate for this method, and return it. Finally, the enumeration implementation searches through the behavior object using reflection and create a list of supported types, but it skips super types whenever one matching method have been added.

```
public IEnumerator<Type> GetEnumerator()
{
    List<Type> tempList = new List<Type>();
    foreach (MethodInfo mthInfo in this.GetType().GetMethods())
    {
        if (mthInfo.Name == ACTION_METHOD_NAME)
        {
            // Only add if a sub-type not allready has been added
            Type mthType = mthInfo.GetParameters()[0].ParameterType;
            foreach (Type type in tempList)
                if (type.IsSubclassOf(mthType))
```

```
                            goto Found;
                    tempList.Add(mthType);
                Found:
                    continue;
                }
            }
            return tempList.GetEnumerator();
        }
```

One could claim that reflection and the rather general approach for adding and invoking behaviors presented above are not very efficient, but for the applications in mind the setup of agents and behaviors are done at initialization. Thus there is no performance overhead of reflection and iteration during runtime, where all activation is handled though lookups in mappings (constant time) and efficient delegates.

Results

As mentioned in the introduction our motivation was primarily based on bad experiences of performance when we implemented a baggage handling system using JADE [Hallenborg, 2006]; a large complex real life system with extensive communication for coordinating the activities. The system has now been re-implemented using the presented agent platform, which fully eliminated the performance problems.

Figure 2 and 3 showed the lack of responsiveness of agents in JADE as the communication increase. Corresponding results for our platform is showed in figure 6.

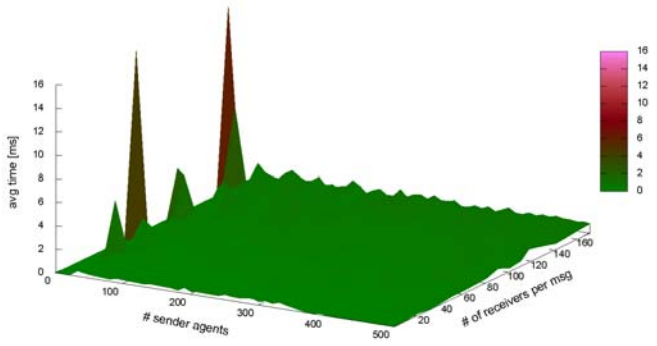

Figure 6: Average message sending time for this agent platform

Apart from a few non-consistent spikes in the case of just one sender agent, the graph clearly illustrates that the average message sending time is constant regardless of the number of sender agents and the number of messages they transmit. The average message time is between 0.5 to 1.0 ms, even for 500 agents each sending 160 messages. A factor 10,000 less than the result in figure 2 for 500 agents sending 80 messages (the highest number possible to generate on the test machine).

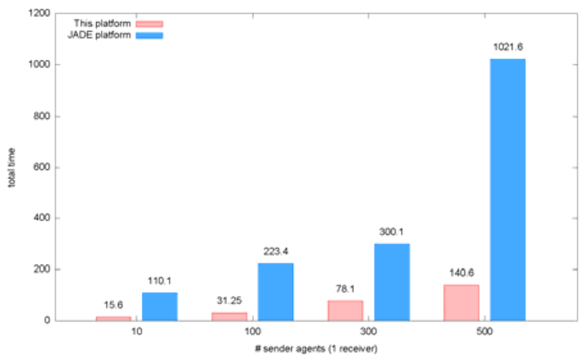

Figure 7: Total time for different number of agents, but only one message sent per agent

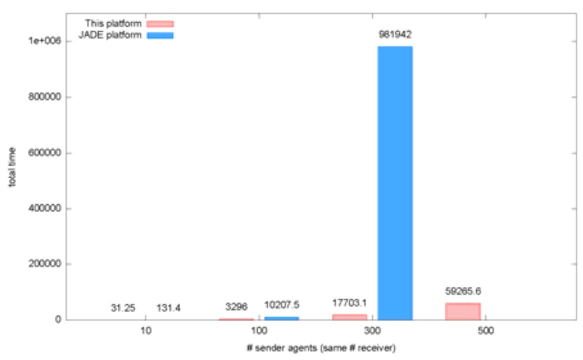

Figure 8: Total for different number of agents, sending the same number of messages per agent

Truthfully, the average message time and responsiveness is just one side of the story. The total computation time of the system is another important factor. Figure 7 and 8 illustrate how our agent platform outperforms JADE.

The results are very clear. Even with only one message sent per agent, where message duplication not should be a problem in JADE, our platform is still a factor 4-7 times faster than JADE. Keep in mind that these implementations are the simplest possible to test the communication model, no advanced encoding and decoding of message content are included, which would slow the JADE solution ever further. In figure 8, where the communication increases dramatically our platform could send and receive the 90,000 message within 18 seconds, but it took more than 16 minutes in JADE. The test machine could not complete the 500 times 500 example in JADE.

Conclusion

We have presented a pattern structure to implement the core communication model of an agent platform. Implementations details that take advantages of special language constructs in C# to efficiently activate and invoke the behaviors of an agent are outlined. The implemented agent platform is compared to agent implementations in JADE, and the presented platform clearly outperforms JADE in all situations, and especially when the communication increase. So for the applications domains in mind (large complex manufacturing and logistics) we have created a very efficient agent platform.

Bibliography

[Weiss, 1999] G.Weiss. Multiagent Systems - A modern approach to distributed artificial intelligence. MIT Press, 1999.

[Jennings, 2000] N.Jennings and M.Wooldridge. Agent-Oriented Software Engineering. In: Handbook of Agent Technology. Ed. J.Bradshaw. AAAI/MIT Press, 2000.

[Searle, 1969] J.R.Searle. Speech Acts. Cambridge University Press, 1969.

[FIPA, 2002] Foundation for Intelligent Physical Agents (FIPA), FIPA Communicative Act Library Specification, 2002.

[Cheong, 2006] C.Cheong and M.Winikoff. Hermes: Designing goal-oriented agent interactions. In: Agent-Oriented Software Engineering VI: 6th Int. Workshop. Ed. J.P.Müller and F.Zambonelli. Lecture Notes in Computer Science, Vol. 3950 Springer Verlag, page 16-27, 2006.

[Braubach, 2007] L.Braubach and A.Pokahr. Goal-Oriented Interaction Protocols. In: Fifth German conference on Multi-Agent System TEchnologieS (MATES-2007), 2007.

[Cabri, 2006] G.Cabri, L.Ferrari, and L.Leonardi. Supporting the Development of Multi-Agent Interactions via Roles. In: Agent-Oriented Software Engineering VI: 6th Int. Workshop. Ed. J.P.Müller and F.Zambonelli. Lecture Notes in Computer Science, Vol. 3950 Springer Verlag, page 154-166, 2006.

[Bellifemine, 2005] F.Bellifemine, F.Bergenti, G.Caire and A.Poggi. JADE - a java agent development framework. Multi-Agent Programming: Languages, Platforms and Applications. Ed. R.Bordini, M.Dastani, J.Dix and A.Seghrouchni. Number 15 in Multiagent Systems, Artificial Societies, and Simulated Organizations. Springer, page 125–148, 2005

[Hallenborg, 2006] K.Hallenborg and Y.Demazeau. Dynamical Control in Large-scale Material Handling Systems through Agent Technology. The 2006 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT-06), HongKong, China, December 18-22, 2006.

Authors' Information

Kasper Hallenborg – Assistant Professor; Maersk McKinney Moller Institute, University of Southern Denmark, Campusvej 55, DK-5230 Odense M, Denmark; e-mail: hallenborg@mmmi.sdu.dk