Extending the JADE Agent Behaviour Model with *JBehaviourTrees Framework*

Iva Bojic[1], Tomislav Lipic[2], Mario Kusek[1], and Gordan Jezic[1]

[1] University of Zagreb, Faculty of Electrical Engineering and Computing
Unska 3, HR-10000, Zagreb, Croatia
{iva.bojic, mario.kusek, gordan.jezic}@fer.hr
[2] Rudjer Boskovic Institute, Centre for Informatics and Computing
Bijenicka 54, HR-10000, Zagreb, Croatia
tlipic@irb.hr

Abstract. Creating modular behaviours in JADE using a traditional method such as the FSM (Finite State Machine) can be a difficult task to achieve. The first issue with FSMs is that they do not allow reusability of logic in different contexts. Secondly, the FSMs do not lend themselves well to concurrency within the execution thread and thus eliminating the possibility for parallel behaviours. Lastly, as the number of states in FSMs becomes increasing large, it becomes difficult to manage them. In this paper we introduce our *JBehaviourTrees Framework* that extends JADE *Behaviours* with BTs (Behaviour Trees) model. BTs are built via the composition of basic tasks increasing the possibility for modularity and code reuse. The proposed approach is verified through a case study concerning a FIPA-Request Interaction Protocol.
Keywords: Behaviour trees, JADE, Finite State Machines, FIPA-Request Interaction Protocol

1 Introduction

In general, each software agent must be autonomous, proactive, reactive and posses some social skills [14]. Specifically, in this paper we are dealing with software agents implemented in JADE (Java Agent DEvelopment Framework), since JADE implements all basic FIPA (Foundation for Intelligent Physical Agents) specifications. In JADE, agents have full control over their internal states and behaviours and the only way for them to communicate with other agents is by sending ACL (Agent Communication Language) messages. Therefore, they have their autonomy, but also by communicating with others, are able to be social. Moreover, agents' proactiveness can be achieved implementing their behaviours as goal-driven [2]. Although, the design choice of JADE was to keep the agent abstraction simple, without requiring explicit representation of goals and mental attitudes, JADE user community provided different solutions [12, 8]. Finally, reactive agents typically are more action oriented as they map their perceptions into actions.

There are numerous methods for building simple, reactive agents based on FSMs (Finite State Machines) [7], HFSMs (Hierarchical Finite State Machines)

[5, 4] and BTs [10, 1] (Behaviour Trees). FSMs are widely used as the main technology when creating the Artificial Intelligence (e.g. in games for nonplayers characters) because of their efficiency, simplicity and expressibly. However, FSMs do not allow reusability of logic in different contexts. Therefore, in order to reuse them, states must be duplicated (causing redundancy) or a great number of complex transitions must be added. Moreover, FSMs suffer from the problem of becoming unmanageable past a certain size, as maintaining the $O(n^2)$ transitions between states becomes an increasingly difficult problem [6].

HFSMs reduce this issue by allowing grouping together a set of states (i.e. super-states) that have common transitions. Different super-states can be then grouped together creating a hierarchy. However, they do not allow reusing of states in different situations because transitions are hard-coded in them. The solution is to use the BTs, which have the same power as HFSMs, but move transitions to external states, so states become self-contained. Every state therefore encapsulates some piece of logic transparently and independently of the context [13].

The main contributions of this paper are summarized below:

- introduction and implementation of BT model into JADE within *JBehaviourTrees Framework*, and;
- its functional evaluation on FIPA-Request IP (Interaction Protocol).

The rest of the paper is organized as follows. Section 2 introduces BTs, while Section 3 presents different types of agents behaviours in JADE and the way how they are used to implement BTs explaining our *JBehaviourTrees Framework* (letter *J* is abbreviation for JADE). Section 4 illustrates FIPA-Request Interaction Protocol case study comparing protocol's standard implementation as FSMs with our implementation of this protocol using *JBehaviourTrees*. Finally, Section 5 concludes the paper and gives an outline for future work.

2 Behaviour Trees

The main building block of BT is a task, instead of state in FSMs and HFSMs. Table 1 shows types of tasks in BTs: *Composite tasks* (i.e. *Selectors*, *Sequences* and *Parallels*), *Decorator task* and *Leaf tasks* (i.e. *Actions* and *Conditions*) [11]. Leaf task performs an *action* or determines if a *condition* has been met. Composite tasks and Decorator are composed of children tasks that can be either composite tasks (i.e. Composites and Decorator) or Leaves.

Execution process in BTs can be described through handling the return status of tasks and their (un)blocking policies (see Table 1). Each task returns a status code indicating success or failure (possibly with error statuses). Leaf tasks return status codes indicating whether they succeeded or failed (e.g. Condition task returns success if condition C is met, and failure otherwise), while status code from Composite tasks depends on their children. Moreover, in order to avoid waiting on some tasks (e.g. while they wait on messages), every single task can be blocked. The *block()* method puts the task in a queue of blocked tasks, while *unblock()* method restarts a blocked task.

Table 1. Types of BTs' tasks

Task	**Symbol**	**Execution process**	
Selector		succeeds if one child succeeds	(un)blocks when its currently active child (un)blocks
Sequence		fails if one child fails	
Parallel		fails if F children fail (Failure policy); succeeds if S children succeed (Success policy)	blocks when all its children block; unblocks as soon as any of its children unblocks
Decorator		manipulates with the return status of its child	(un)blocks itself and its children
Action		fails if it cannot complete the action	(un)blocks itself
Condition		succeeds if condition C is met	

2.1 Composite tasks

The root of BT is basically top-level task that can be then decomposed recursively into sub-tasks accomplished with simpler tasks called *Composite tasks*. Composite tasks are tasks with one or more children, which act as decision nodes within the tree determining which child task to execute. Their execution depends on the execution of their children. When child task is completed and has returned its status code, the Composite decides whether to continue through its other children or whether to stop and return a value.

Selector. Selectors are used to choose the first child that is successful. A Selector will return immediately with a success status code when one of its children ends successfully. Each Selector can have its own criteria for selecting the child task:

- *Probability selector* - chooses a child task based on a probability (random or specified by user), and;
- *Priority selector* - chooses a child task based on the order of children (first child is most important).

Sequence. Sequence sequentially executes all its children in order. It will return immediately with a failure status code when one of its children fails. As long as its children are succeeding, it will keep going. If it runs out of children, it will return success.

Parallel. Parallel task supports execution of concurrent tasks, running all of its children at the same time. A Parallel task has a parallel policy: *Failure policy* or *Success policy* (see Table 1). Symbol for Parallel task has letter F indicating how many children have to fail, so Parallel task would fail. Analogously, letter S detonates the number of children that have to succeed in order for Parallel task to be successful.

2.2 Decorator task

The name *decorator* is taken from object-oriented software engineering. The decorator pattern refers to a class that wraps another class, modifying its behaviour. Decorator in BTs is a type of task that has one single child task and modifies its behaviour. It takes the original task and adds decorations, i.e. new features. For instance, Decorator can be used to: limit the number of times a task can be run, prevent task from firing too often with a timer or restrict the number of tasks running simultaneously.

2.3 Leaf tasks

Leaf tasks are tasks with no children: *Actions* and *Conditions*. An Action task performs an action, while Condition task determines if a condition C has been met. By separating Leaf tasks into Actions and Conditions, BTs become more easily understood and adaptable, extending behaviour modularity and functionality. Therefore, Composite tasks can be used in a way that allows them to act as pre-conditions or assumptions for other tasks.

3 *JBehaviourTrees Framework:* Implementing BTs into JADE

In JADE, *Behaviour* class is an abstract class for modelling agent behaviours. It is responsible for the basic behaviour scheduling and state transitions, such as starting, running, blocking, restarting and terminating JADE's behaviours. Since *Behaviour* class, along with its subclasses (e.g. *CompositeBehaviour*, *SimpleBehaviour* and *WrapperBehaviour*) provide whole functionality and control of behaviours execution process, we explain how we used that mechanisms in our *JBehaviourTrees Framework*.

3.1 Implementing BTs tasks functionality

Our *JBehaviourTrees Framework* consist of six classes: *SelectorTask*, *SequenceTask*, *ParallelTask*, *DecoratorTask*, *ActionTask* and *ConditionTask* classes. We used *CompositeBehaviour* class to implement set of Composite tasks, *WrapperBehaviour* class is extended in classes that implement functionality of *DecoratorTask*, and finally, Leaf tasks are implemented as *SimpleBehaviours*.

Composite tasks. *SelectorTask* and *SequenceTask* extend *SerialBehaviour*, a composite behaviour with serial children scheduling. They both implement *checkTermination()* method that checks whether to terminate. *SelectorTask* terminates when one child successes (returning success) or ends when all children are executed and none of them succeed (returning failure), while *SequenceTask* terminates when one child fails (returning failure) or ends when all children are executed and none of them failed (returning success). *ParallelTask* is implemented extending *ParallelBehaviour* class, a composite behaviour with concurrent children scheduling. *ParallelTask* terminates (and returns success) when a particular condition on its sub-behaviours is met (i.e. when all children are done, N children are done or one child is done). Otherwise it ends returning failure.

Decorator task. *DecoratorTask* is implemented extending *WrapperBehaviour* class. This class allows modifying on the fly the way an existing behaviour object works, and therefore provides a good decorator design pattern for BTs Decorator task implementation.

Leaf tasks. Both, *ActionTask* and *ConditionTask* are implemented as *OneShotBehaviours*, that completes immediately after its *action()* method is executed exactly one time. When some condition must be checked repeatability, or some action executed more than once, *DecoratorTask* can be used to determinate the number of times leaf task must be run.

3.2 Controlling BTs tasks execution process

In BTs, the route from the top level to each leaf represents one course of action, and the execution of BT evaluates those courses of action in a left-to-right manner performing a depth-first search. The control of this execution process can be achieved using JADE's behaviours scheduling, control handling and (un)blocking mechanisms.

Scheduling. If an agent in JADE, has more independent behaviours (added using *addBehaviour()* method in base *Agent* class), their execution is controlled by JADE's scheduler. Scheduler, implemented by the base *Agent* class, maintains a set of active behaviours that are then executed in a round-robin manner. Behaviours, that are not independent behaviours, but sub-behaviours of *CompositeBehaviour* (e.g. like tasks in BTs), have their own scheduling policies realized through "*control handling mechanisms*".

Control handling mechanisms. Composite and Decorator tasks in BTs are used to control the flow within the tree, while Leaf tasks execute code returning success or failure. In all cases, except when the task is currently running, control is passed back to the parent task. The parent task then handles the given return status of its child passing it up to its parent.

In JADE, the *CompositeBehaviour* class provides only a common interface for children scheduling, but does not define any scheduling policy. This means that the actual operations performed by executing this behaviour are not defined in the behaviour itself, but inside its children. Scheduling policies, defined by its subclasses (*SerialBehaviour* and *ParallelBehaviour* classes) are already explained in Section 3.1.

As for the returning codes, the *Behaviour* class provides a place-holder method named *onEnd()*. This method must be overridden returning an *int value* that represents a termination value for that behaviour. This int value denotes either that behaviour has completed its task successfully, or that it was terminated after some error had happened.

(Un)blocking mechanisms. In order to maintain the list of active tasks ready for execution, each task must be classified as *running*, *blocked* or *terminated*. Task is in runnable phase until it is blocked or terminated. When blocked, it enters in inactive mode and is not in list of active tasks.

In JADE, a behaviour is active when executing its *action()* method. By using *block()* method, behaviour can be put in a queue of blocked behaviours, and can be unblocked when *restart()* method is called, while *handle()* method handles block/restart events in behaviours through *RunnableChangedEvent* class. *RunnableChangedEvent* class is used to notify interested behaviours when certain behaviour changes its runnable state sending this information to behaviour's parent (upward notification) or to behaviour's children (downward notification). Finally, behaviour is terminated when its *done()* method returns *true*.

3.3 Achieving BTs task collaboration

Agents in JADE can communicate transparently sending ACL messages. Their communication is based on an asynchronous message passing paradigm. But, for inner-agent communication (i.e. collaboration), between behaviours within one composite behaviour (i.e. *CompositeBehaviour* class), JADE provides private data store for each behaviour through *DataStore* class. In this way a blackboard architecture for *CompositeBehaviours* and their children is realized and it is ground for data sharing among our JADE BTs tasks.

The blackboard is a useful external data store for exchanging data between behaviours (i.e. tasks). When agents communicate by writing and reading from the blackboard, they are independent from each other, and that is a better way than calling methods. Moreover, having all behaviours communicate in this way, allows a usage of existing data in novel ways, making it quicker to extend the functionality of implementation.

4 Case study: FIPA-Request Interaction Protocol

FIPA-ACL [3] specifies a collection of communicative acts where semantics of each act is specified in terms of a set of feasibility preconditions and a set of

REs (Rational Effects). Interaction protocols specify agent communication by defining sequences of speech acts that can be used for specific type of interactions between agents. In this section we explain the FIPA-Request IP.

The FIPA-Request IP [3] allows one agent (i.e. *Initiator*) to request another (i.e. *Participant*) to perform some action. The Participant processes the request and makes a decision whether to accept it or refuse it. Once the request has been agreed upon, then the Participant must communicate either: a *failure*, an *inform-done* or an *inform-result*.

4.1 FIPA-Request IP implemented as FSM

In JADE, FIPA-Request IP is implemented with two classes: *AchieveREInitiator* and *AchieveREResponder* (see Figure 1). *AchieveREInitiator* class extends *Initiator* class that is implemented as *FSMBehaviour* (implements FSM model into JADE), while *AchieveREResponder* directly extends *FSMBehaviour* class.

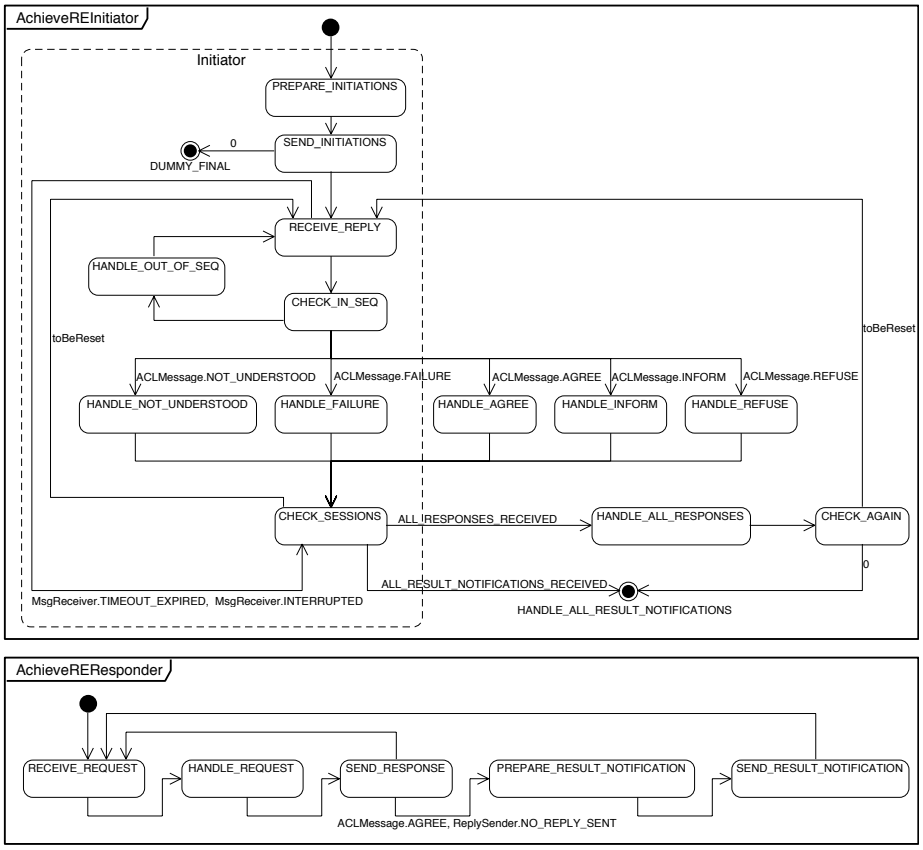

Fig. 1. FIPA-Request IP implemented as FSM

4.2 FIPA-Request IP implemented as BT

Figure 2 shows our implementation of FIPA-Request IP using *JBehaviourTrees Framework*. *AchieveREInitiatorBT* class implements the same functionality as *Achi-eveREInitiator* class. It is realized using one *SequentialTask* where an agent forms one or more requests, sends them, and then waits on responses until all are received, or a time-out occurs. Waiting on responses is implemented within one *ParallelTask*, while *DecoratorTask* counts how many messages are received. After one message is received, we use *SelectorTask* in order to determinate type of the response (e.g. failure, not understood, agree). *AchieveREResponderBT* can be used interchangeably with *AchieveREResponder*. It is implemented using one *DecoratorTask* that never ends, and then through one *SequentialTask* is achieved functionality of receiving request and sending response.

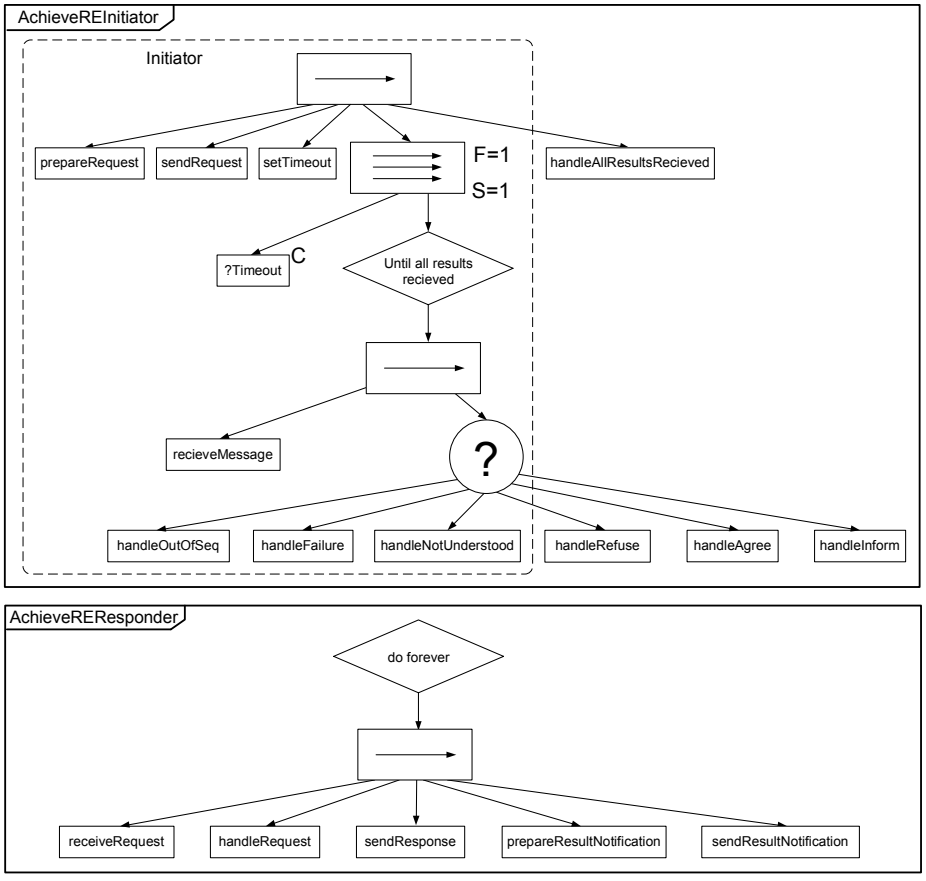

Fig. 2. FIPA-Request IP implemented as BT

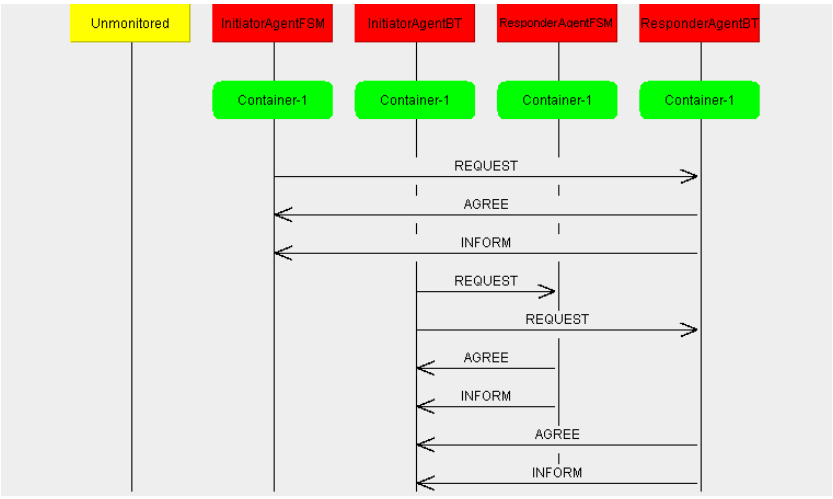

Fig. 3. Message exchange in our version of JADE Sniffer [9]

4.3 Evaluation results

In order to make functional evaluation of BT implementation in case study, we made three testing sequences: *InitiatorAgentFSM* interacting with *ResponderAgentBT*, *InitiatorAgentFSM* interacting with *ResponderAgentBT* and vice versa. Figure 3 shows messages exchanged in all three scenarios. This verifies the functionality of BT approach and compatibility with build–in FSM implementation.

5 Conclusion

In this paper, we made functional prototype of FIPA-Request Interaction Protocol using proposed *JBehaviourTrees Framework* that provides BTs model for JADE behaviours. We showed that both the FSM implementation and implementation based on BTs can be used interchangeably.

Advantage of our approach is that BT model provides better code reusability. By creating a few simple BTs almost any desired functionality can be achieved by linking trees in different ways or extending the functionality through decorators. This form of creating complex behaviours is more favourable to FSM or even HFSMs due to the large amount of freedom BTs intrinsically support.

However, sometimes it is hard to build BTs that must implement state-machine-like behaviours since that can be only done by creating unintuitive trees. Therefore, for the future work we plan to build a hybrid system, where agents will have multiple BTs and state machines will be used to determine which BT they are currently running.

Acknowledgments. The authors acknowledge the support of research project "Content Delivery and Mobility of Users and Services in New Generation Networks" (036-0362027-1639), funded by the Ministry of Science, Education and Sports of the Republic of Croatia.

References

1. AiGameDev-site: http://aigamedev.com/insider/presentations/behavior-trees
2. Bellifemine, F., Caire, G., Poggi, A., Rimassa, G.: JADE: A software framework for developing multi-agent applications. Lessons learned. Information and Software Technology 50, 10–21 (2008)
3. FIPA: http://www.fipa.org
4. Fortino, G., Rango, F., Russo, W.: Statecharts-Based JADE Agents and Tools for Engineering Multi-Agent Systems. In: Setchi, R., et al. (eds.) Knowledge-Based and Intelligent Information and Engineering Systems, Lecture Notes in Computer Science, vol. 6276, pp. 240–250. Springer, Berlin, Heidelberg (2010)
5. Griss, M.L., Fonseca, S., Cowan, D., Kessler, R.: Using UML state machine models for more precise and flexible JADE agent behaviors. In: Proceedings of the Third International Conference on Agent-oriented Software Engineering. pp. 113–125. Springer, Berlin, Heidelberg (2003)
6. Heckel, F.W.P., Youngblood, G.M., Ketkar, N.S.: Representational Complexity of Reactive Agents. In: Proceedings of the IEEE Conference on Computational Intelligence and Games. pp. 257–264. IEEE (2010)
7. JADE-FSM-Builder: http://www.sicnet.it/jade-fsm-builder
8. Jurasovic, K., Jezic, G., Kusek, M.: Using BDI agents for automated software deployment in next generation networks. In: Proceedings of the Eleventh International Conference on Software Engineering and Applications. pp. 423–428. ACTA Press (2007)
9. Kusek, M., Jezic, G.: Extending UML sequence diagrams to model agent mobility. In: Proceedings of the Seventh International Conference on Agent-oriented Software Engineering. pp. 51–63. Springer, Berlin, Heidelberg (2007)
10. Lim, C.U., Baumgarten, R., Colton, S.: Evolving Behaviour Trees for the Commercial Game DEFCON. In: Chio, D., et al. (eds.) Applications of Evolutionary Computation, Lecture Notes in Computer Science, vol. 6024, pp. 100–110. Springer, Berlin, Heidelberg (2010)
11. Millington, I.: Artificial Intelligence for Games. Morgan Kaufmann Publishers, San Francisco, USA (2009)
12. Poggi, A.: Developing multi-user online games with agents. WSEAS Transactions on Computers 7, 1240–1249 (2008)
13. Puga, G.F., Gómez-Martín, M.A., Díaz-Agudo, B., González-Calero, P.A.: Dynamic Expansion of Behaviour Trees. In: Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference. pp. 36–41. The AAAI Press (2008)
14. Wooldridge, M., Jennings, N.R.: Intelligent Agents: Theory and Practice. Knowledge Engineering Review 10, 115–152 (1995)