The Use of Design Patterns for the Development of Multi-Agent Systems

Kostas Kostiadis, Matthew Hunter, Huosheng Hu

Department of Computer Science, University of Essex,
Wivenhoe Park, Colchester CO4 3SQ, UK
Email: kkosti@essex.ac.uk, mchunt@essex.ac.uk, hhu@essex.ac.uk

Abstract. *Developers of AI software are normally faced with design challenges involving robustness, efficiency, and extensibility. Most of these challenges at a higher level are independent of the application-specific requirements. Although design patterns have been successfully adopted to tackle these issues, they are rarely documented. Consequently this knowledge remains hidden in the minds of developers or buried within complex system source code. The primary contribution of this paper is to describe an abstract design methodology that can be applied in many (single or) multi-agent systems. The paper mainly illustrates how design patterns can ease the development and increase the efficiency of such systems. As an example, the Essex Wizards multi-agent system is presented which won the third prize in the RoboCup'99 simulator league competition.*

Keywords: Multi-agent systems, Agent Architecture, Design Patterns, Robot World Cup.

1. Introduction

As research on AI and multi-agent systems (MAS) advances, and the complexity of development of such systems grows rapidly, developers are normally faced with software design challenges involving robustness, efficiency, and extensibility. Most of these challenges at a higher level are independent of the application specific requirements. Although there are many similarities in the different forms of AI software in general, the focus of this paper is on MAS software, which has both *inherent* and *accidental* complexities.

Inherent complexities stem from fundamental challenges of developing MAS software. For instance, most MAS are prohibitively complex for handcrafted decision making approaches. In such cases the use of machine learning techniques can greatly improve the decision making process of an agent [Kostiadis et al., 1999]. In addition to that, co-operation between agents is also an important challenge in MAS [Tambe, 1997]. Co-operation further introduces the challenges of communication [Stone, 1999] and role variation [Coradeschi et al., 1998] between agents. Finally, in adversarial domains, an additional challenge is that of opponent modelling [Riley, 1999].

Accidental complexities associated with MAS software come from limitations with conventional tools and techniques. For example, low-level application programming interfaces (APIs) like Sockets are tedious and error prone. Another source of accidental complexity arises from the widespread use of algorithmic design to develop MAS software. Algorithmic design yields non-extensible software architectures that cannot be customised rapidly to meet changing application requirements [Schmidt, 1997]. Object-oriented (OO) techniques provide principles, methods, and tools that significantly reduce the complexity and cost of developing software [Fayad et al., 1997]. The main benefits of OO arise from the emphasis on modularity, reusability and extensibility. A design pattern represents a recurring solution to a software development problem within a particular context [Gamma et al., 1995]. Patterns capture abstract designs and software architectures in a systematic format that can be readily comprehended by developers [Schmidt, 1997].

In this paper both OO techniques and design patterns have been adopted to design MAS software for the robotic soccer team, Essex Wizards, in order to reduce the development time and achieve a more stable and efficient system. The creation of the robotic soccer, the robot world

cup initiative (RoboCup), is an attempt to foster AI and intelligent robotics research by providing a standard problem where wide range of technologies can be integrated and examined [Kitano, 1997][Kitano et al. 1997]. Some of the fields covered include multi-agent collaboration, strategy acquisition, real-time planning and reasoning, sensor fusion, strategic decision making, intelligent robot control, and machine learning. Such a complex, real-time environment, with presence of teammates and adversaries, provides numerous challenges for the development of a successful multi-agent system.

The remainder of this paper is organised as follows: Section 2 briefly describes the difficulties in designing a multi-agent system and also shows the proposed agent architecture. Following this, section 3 illustrates the multi threaded mediator pattern that has been used. Finally experimental results and conclusions are presented in section 4.

2. Designing a Multi-Agent System

An agent is any system that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors [Russell et al., 1995]. A multi-agent system is a collection (or team) of such agents. In order to create a coherent team of agents, the entire design process and the agent architecture must be built with the team in mind. "Collaboration must be designed into systems from the start; it cannot be patched on [Grosz, 1996]." Also during the design process the issues of robustness, reusability, and extensibility must be considered. It has been shown [Gamma et al., 1995] that design patterns can help the potential developer deal with a number of problems including decomposing a system into objects, determining their granularity, and specifying their interfaces and implementation.

Before explaining how design patterns have been used in the Essex Wizards multi-agent system, a brief description of the proposed agent architecture is required. A more detailed description of the overall agent can be found in [Kostiadis et al., 1999b]. As mentioned earlier, the robotic soccer domain provides a real-time environment where each agent is required to have bounded response times in order to achieve

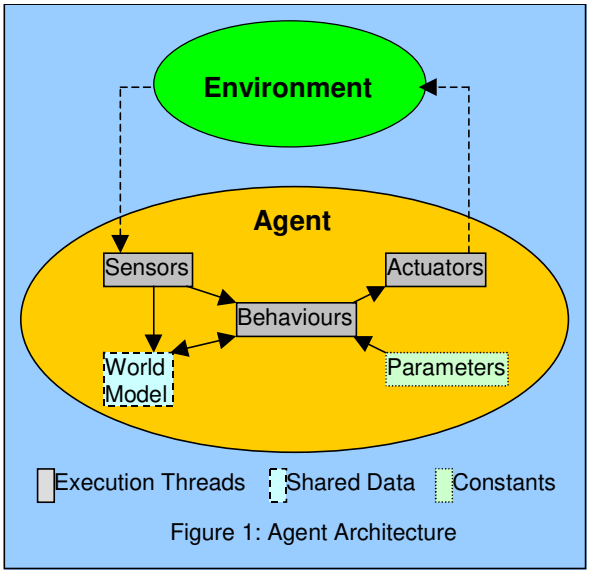


Figure 1: Agent Architecture

acceptable levels of performance. To achieve such performance the Essex Wizards use a multi-threaded approach where the 3 stages of the "sense-think-act" cycle are executed concurrently. It has been shown that this form of concurrent execution guarantees bounded response times [Kostiadis et al., 1999b] and therefore improves the overall performance of the agent.

At an abstract level, the agent contains 5 different modules: the agent's sensors, the actuators, a set of predefined parameters that affect the thinking process, the decision module, and a representation of the world model, as shown in figure 1. Each agent receives information regarding the environment from its sensors, decides what the best action is depending on the current state, and sends this action through its actuators back to the environment. The solid arrows represent dependence relationships between these modules.

More specifically, "sensors" are responsible for receiving the newest input from the environment, and updating the agent's "world model". The newest sensory input also affects the "behaviours" module. The "world model" holds a probabilistic representation of the whole environment. This is of significant importance when there is hidden state and the sensors cannot provide a complete representation of the current state of the environment. "Parameters" hold information regarding the agent's role in the team, or certain features specific to a particular agent (stamina, home position, etc.). These 3 modules directly affect the decision process and hence the

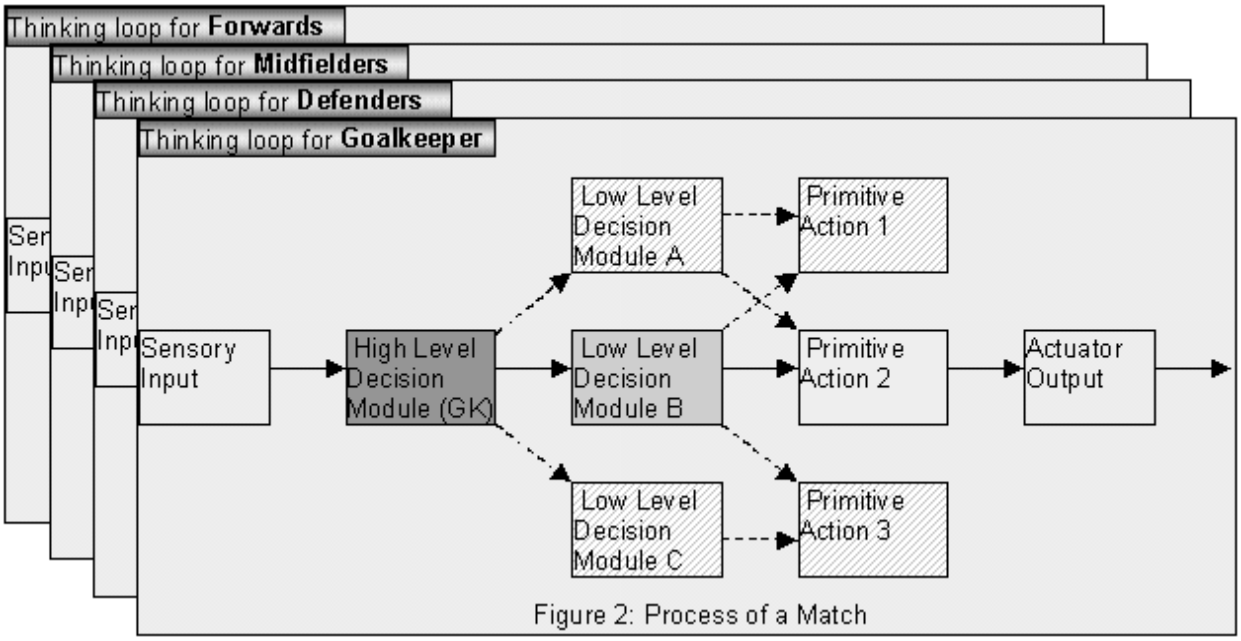


Figure 2: Process of a Match

"behaviours" module of the agent. The "behaviours" module is responsible for deciding the best course of action and sending that action to the actuators. The actuators in turn send this action back to the environment (figure 2). Due to perception and action being asynchronous in the RoboCup domain, after an action has been sent, the "behaviours" module internally updates the world model using the predicted effects of that action. This helps the agent maintain an up-to-date representation of the environment in case sensory data is delayed or missing.

In addition to the complexity of building each module, there are normally multiple people involved in the implementation. Under these circumstances, integration and testing becomes a challenging task. The proposed design should be modular, highly flexible and scalable. All the modules described above, are represented by various objects, in a fully object oriented design. Due to the tight coupling of the independent components within the agent, the design should allow easy modification, addition and removal of various modules, and provide a flexible interface for the inter-communication between objects. Finding an acceptable solution to this problem is not a trivial task.

3. Multi-Threaded Mediator Pattern

The well-known Mediator design pattern [Gamma et al., 1995] has been used to tackle this problem. The idea behind the Mediator pattern is to define a module that encapsulates how sets of modules interact. Mediator promotes loose coupling and modules within the agent can communicate without explicitly referring to each other. Given the multi-threaded nature of the Essex Wizards team, an extension to the known Mediator pattern had to be found. Shared data like the "world model" had to be protected from accidental reads/writes and some communication links had to be synchronised. This section briefly describes the Mediator pattern and presents the extensions added in order to build a design pattern that can operate in a multi-threaded environment.

Breaking an agent into individual modules encourages the distribution of behaviour among these modules. However, such distribution can result in a structure with many connections between the various modules. Although partitioning a system into objects/modules normally enhances reusability, having too many interconnections tend to reduce it again. Numerous inter-connections make it more difficult for a module to work without the support of others. In addition to that, making significant changes to the overall behaviour of the system becomes unnecessarily difficult, since behaviour is distributed among many modules.

These problems can be avoided by encapsulating collective behaviour in a separate mediator module as shown in figure 3. The mediator controls and co-ordinates the interactions of the various modules within the agent. The mediator

acts as a communication hub between the modules, keeping them from referring to each other, and hence reducing the number of inter-connections between them. Each module is only required to know its mediator and all communication with other modules is done indirectly through this channel. An additional benefit comes from the localisation of the overall behaviour in one module. The system can easily change its behaviour by modifying or replacing just the mediator module.

Although the mediator design pattern promotes loose coupling and reduces inter-connections between modules, it also tends to centralise the overall control of the system. In other words there is a trade-off between complexity of module interaction and complexity in the mediator. Since one object is used to encapsulate communication between all modules, the mediator can become more complex than any individual component within the agent. The potential developer must be careful in order to prevent the mediator itself from becoming monolithic and hard to maintain.

As mentioned earlier, the Essex Wizards multi-agent system is based on a multi-threaded implementation. The traditional mediator pattern is not capable of behaving as intended in such an environment. Assume for instance that the "sensors" module has just received new information and started updating the world model. While the update is taking place, the "behaviours" module, which operates from a different thread, might try and read the contents of the world model. Since the update has not yet been completed, the world model contains up-to-date information for part of the environment, and older information for the rest of the environment. In most cases this will result in sub-optimal or even incorrect actions being taken. Needless to say a synchronisation protocol needs to be integrated into the current mediator pattern.

The most common and general way to synchronise data access between threads is to ensure that all read/write operations on shared data are mutually exclusive. The POSIX threads API that has been used in the Essex Wizards implementation provides mutual exclusion using a special form of Edsger Dijkstra's semaphore, called a mutex

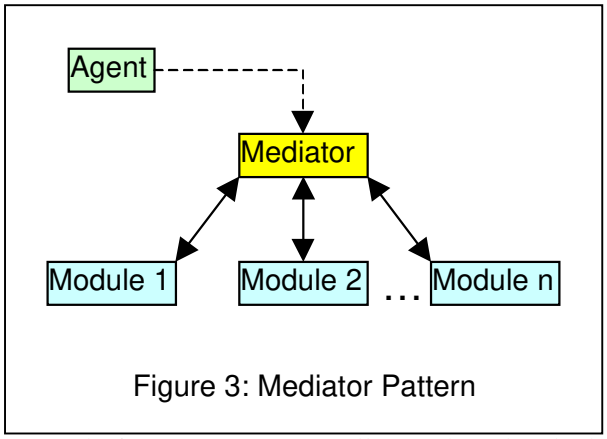


Figure 3: Mediator Pattern

[Butenhof, 1997]. Mutexes allow a thread to lock specific data items so that no other thread can operate on them at the same time. In the example mentioned above the "sensors" module would have to lock the contents of the world model while the update was taking place. This would result to the "behaviours" module being put to a halt until the update was finished and the sensors unlocked the specified mutex. A schematic representation of this can be seen in figure 4.

To enable the mediator pattern to function correctly in a multi-threaded environment, similar synchronisation had to be integrated into the various communication channels between the mediator and specified modules. There is always a trade-off between successful synchronisation protocols and real-time responsiveness. Since various threads can be halted, it is important to lock mutexes for the minimum time required for a read or write operation to complete, and this must occur only when absolutely necessary. In the current implementation, synchronisation takes place mainly within the mediator, just before control is passed to a module. This way, operations within modules become atomic, and synchronisation is transparent to the individual components of the agent. Needless to say there are cases where the mediator is not involved in an operation, yet synchronisation between shared data needs to be applied. These cases are rare and occur when certain operations need to take place within a single module without the need for communication with other modules. In such cases the mediator is not directly involved and synchronisation can only take place within the given module.

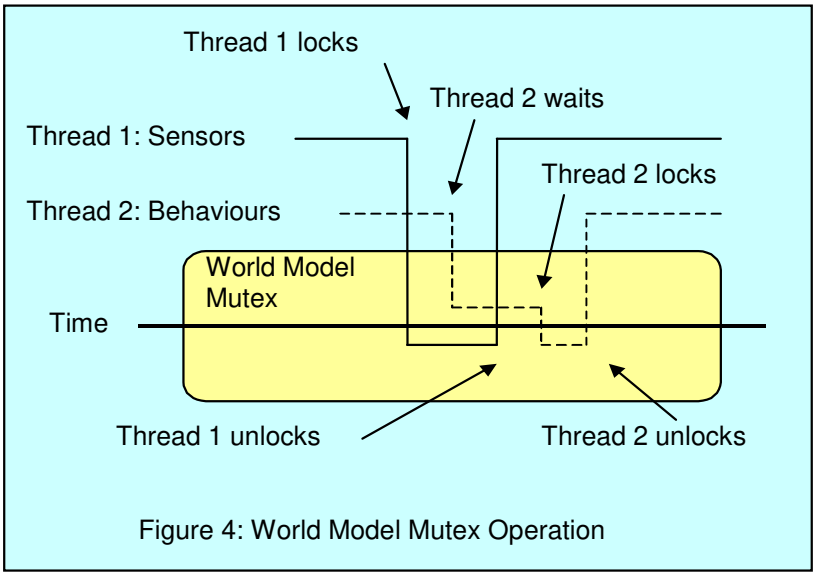


Figure 4: World Model Mutex Operation

4. Results & Conclusions

In recent years there has been significant progress in the fields of AI and multi-agent systems. As hardware becomes better and faster, more advanced and complex systems are being developed. Following this growth in hardware, AI software has also experienced rapid growth in terms of complexity, functionality and performance. However, most developers of AI software face similar design challenges involving robustness, efficiency, and extensibility. Most of these challenges at a higher level are independent of the application specific requirements.

In this paper it has been shown how object-oriented techniques and design patterns have been used in order to reduce the complexity and cost of developing multi-agent system software. The main benefits of this approach arise from the emphasis on modularity, reusability, and extensibility. Although numerous patterns have been used in the implementation of the Essex Wizards multi-agent system, this paper illustrated how the mediator pattern can be used in a multi-threaded environment. Using the mediator pattern results in reduction of the inter-connections between the various modules within an agent and also provides a simpler and highly flexible communication interface between them. Although only the multi-threaded mediator has been presented in this paper, multi-agent system development can be enhanced by a variety of other design patterns.

To fully utilise the benefits of object-oriented techniques and design patterns a very complex simulated environment was chosen, the robotic soccer domain. The proposed architecture and design have been successfully implemented into the Essex Wizards multi-agent system. The Essex Wizards participated in the RoboCup '99 competition, an international event with over 30 participants from universities around the world. The Wizards achieved a combined score of 59-9 over their opponents and finished in third place.

Acknowledgements

We would like to thank the University of Essex for the financial support to this project by providing the Research Promotion Fund DDP940.

References

[Butenhof, 1997] Butenhof D. R., *Programming with POSIX Threads*, Harlow, Addison-Wesley, 1997

[Coradeschi et al., 1998] Coradeschi S, and Karlsson L., *A Role-Based Decision-Mechanism for Teams of Reactive and Coordinating Agents*, "RoboCup-97: The First Robot World Cup Soccer Games and Conferences", Springer Verlag Lecture Notes in AI (LNAI), 1998.

[Fayad et al., 1997] Fayad M. E. and Schmidt D. C., *Object-Oriented Application Frameworks*, Communications of the ACM, vol. 40, October 1997.

[Gamma et al., 1995] Gamma E., Helm R., Johnson R., and Vlissides J., *Design Patterns: Elements of Reusable Object Oriented Software*, Reading, MA: Addison Wesley, 1995.

[Grosz, 1996] Grosz B., *Collaborative Systems*, AI Magazine, 17(2):67-85, 1996.

[Kitano, 1997] Kitano H., *RoboCup: The Robot World Cup Initiative*, In proceedings of The 1st Int. Conf. on Autonomous Agent (Agents-97), Marina del Ray, The ACM Press, 1997.

[Kitano et al., 1997] Kitano H., Tambe M., Stone P., Veloso M., Coradeschi S., Osawa E., Matsubara H., Noda I., and Asada M., *The RoboCup Synthetic Agent Challenge'97*, Proc. of International Joint Conference on Artificial Intelligence (IJCAI), 1997.

[Kostiadis et al., 1999] Kostiadis K. and Hu H., *Reinforcement Learning and Co-operation in a Simulated Multi-agent System,* Proc. of International Conference on Intelligent Robots and Systems (IROS), Korea, October 1999.

[Kostiadis et al., 1999b] Kostiadis K. and Hu H., *A multi-threaded approach to simulated soccer agents for the RoboCup competition,* International Joint Conference on Artificial Intelligence (IJCAI): workshop on RoboCup, 1999.

[Riley, 1999] Riley P., *Classifying Adversarial Behaviours in a Dynamic Inaccessible Multi-Agent Environment*, undergraduate senior thesis, Carnegie Mellon University, 1999.

[Russell et al., 1995] Russell S. and Norvig P., *Artificial Intelligence: A Modern Approach*, Prentice Hall Series in AI, New Jersey, Prentice Hall, 1995.

[Schmidt, 1997] Schmidt D., *Handbook of Programming Languages*, vol. 1, edited by Peter Salus, MacMillan Computer Publishing, 1997.

[Stone, 1999] Stone P. and Veloso M., *Task Decomposition, Dynamic Role Assignment, and Low-Bandwidth Communication for Real-Time Strategic Teamwork*, to appear in Journal of Artificial Intelligence, 1999.

[Tambe, 1997] Tambe M., *Towards Flexible Teamwork,* Journal of Artificial Intelligence Research, vol. 7, pp. 83-124, 1997.