Design Diagrams for Multi-Agent Systems

Simon Lynch
Intelligent Systems Group
University of Teesside
s.c.lynch@tees.ac.uk

Keerthi Rajendran
Computing & Information Systems
Botswana Accountancy College
keerthir@bac.ac.bw

Keywords: POP-II.B. design, POP-IV.A. multi-agent systems

Abstract

Advanced approaches to the construction of software systems can present difficulties to learners. This is true for multi-agent systems (MAS) which exhibit concurrency, non-determinacy of structure and composition and sometimes emergent behavior characteristics. Additional barriers exist for learners in MAS because mainstream MAS technology is young and design methodologies are still evolving.

This paper examines Agent UML - a set of proposed extensions to UML to facilitate MAS design. The paper highlights limitations in Agent UML's ability to accurately describe all aspects of MASs and suggests an additional diagrammatic technique to address these limitations. The additional methodology is intended to complement, rather than replace, those that already form the basis of Agent UML.

Motivation

We are interested in teaching students about MAS as a strategy for software development to enable them to contribute their skills to ongoing development of systems used to support clinicians involved in performing minor surgery. The rationale for using MAS in clinical environments is outlined below. Serious MAS development is still primarily a research exercise or, at the very least, requires considerable time and expertise. Students and even recent graduates face a steep learning curve on entry to practical MAS development. We believe this is partly due to the lack of development platforms containing *off the shelf* agents and also due to the lack of accessible design tools.

This paper considers the need for MAS in clinical environments and the re-use benefits offered by agent technologies in general. It summarises the current state of *Agent UML* (AUML), an evolving standard, highlighting its short-comings and suggesting complimentary diagrammatic techniques which can be used in combination with existing AUML notations. These aim to benefit all MAS designers and MAS novices in particular.

Learning Issues

MAS are distributed and by definition processing is spread across multiple agents. Each agent typically runs in its own process/thread so the processing is implicitly concurrent. Many systems allow agents to join and leave at run-time creating a non-determinacy of structure and composition. Additionally agents cooperate, forming small problem solving communities which sometimes exhibit characteristics of emergent behavior. These factors make their run-time structure and performance hard to predict and make their design challenging.

Students often have difficulties with initial exposure to concurrency issues and studies have noted that learners find distributed control systems more complex to understand that their hierarchical counterparts (Vidal & Buhler 2002; Resnick 1994). Others note learner difficulties with event driven software because the path through program code is distributed across multiple methods (in some languages) and is dependant on the nature of the external events which occur (Lynch & Rajendran 2003a; Petitpierre 2002). Learners face particular barriers if they need to understand the run-time programming environment and/or the program architecture if these are not immediately apparent from the code and if they affect execution of the code.

All these issues present barriers to learners of MAS technology so the availability of suitable design tools is particularly important. The following sections outline the benefits offered by MAS, concentrating on re-use in particular, then discuss Agent UML (AUML) as a possible design methodology.

The Need For Multi-Agent Systems

Clinical systems are unusual because various constraints typically affect computer use in environments where surgery takes place. Possible modes of human-computer interaction are limited for different reasons in different working environments - in a surgical environment clinicians are often unable to use hand operated input devices. Firstly the nature of the medical tasks they perform often requires considerable manipulative dexterity involving both hands - leaving no hands free to operate a mouse or keyboard. Secondly, with surgical procedures that do not consistently require two handed operation there is often a hygiene issue that prevents clinicians using devices, like a mouse or keyboard, which cannot be easily sterilised between the examination of different patients.

Despite these constraints it is often a requirement that data is captured during examinations because delays in capturing or transcribing clinical data can lead to inaccurate reporting. Clinicians, like other users, have a greater likelihood of forgetting or *mis-remembering* data if it is not recorded immediately (Barker *et al* 1999). For obvious reasons this can compromise the integrity of clinical records in ways which damage patient care.

Studies recognise that keyboard and mouse driven interface are not always appropriate (ibid.) and that speech input (an obvious alternative to consider) is not universally appropriate either (Potamianos 1999; Bernsen & Dybkjær 1998). Some suggest using multi-modal interfaces for various types of task (Bekker *et al* 1995; Damper & Wood 1995; Oviatt *et al* 1997) but multi-modal dialog introduces specific problems for system developers and suitable architectures for multi-modal dialog systems are complex (Potamianos 1999).

Since the first experimental multi-modal system "*Put that there*" (Bolt 1980) researchers have recognised some fundamental difficulties with multi-modal input. One such difficulty relates to how and when to fuse input from different sources. Other difficulties are concerned with how to structure the internal architecture of a multi-modal system. The architecture must support systems which cope with either early fusion of data from input modalities, late fusion (perhaps after some semantic processing in the case of speech input) or some hybrid of both types. Some modules associated with multi-modal dialog systems (those associated with language processing for example) are considered to be part of the domain of *Intelligent Systems* and are significant software systems in their own right. Decisions about the *level of intelligence* of components and the style of component-component interaction complicate design choices as do other issues involved with spoken dialog systems: turn taking, user interruption/contradiction of spoken input, dialog management, user modeling, etc. Additionally the build time and level of previous experience required to construct multi-modal systems is considerable. This means that, however desirable a multi-modal approach may be, in many cases commercial factors can make them non-viable.

Authors have noted that, like other areas of computing, "[i]n terms of system architecture, dialog systems have evolved from monolithic, hard-coded system design to a *modular* architecture of re-useable and programmable building blocks" (Potamianos 1999). In the case of multimodal dialog systems these building blocks often exist as agents (Djenidi 2002a) since components typically need to run simultaneously (in order to accept simultaneous input from different sources) and sometimes need to interrupt each other (for example when user interrupts a speech output agent). Recognising that multi-modal system developers do not yet follow any generic architectures for their systems, Djenidi *et al* have identified and examined different architectures that have some merit. All of these architectures are multi-agent in structure (Djenidi 2002b).

While a multi-agent system (MAS) is a suitable architecture for a multi-modal dialog system it does not immediately solve problems of system construction since MASs are themselves complex. According to Sycara (2001) "the development of a MAS is extremely challenging" and other authors agree (Gasser 2001). The opportunities for re-use which could exist with MASs can only be exploited if suitable (re-useable) MAS designs are in the public domain along with collections of predefined agents which can either be used directly or easily specialised for current needs. In summary what is required is a set of design patterns for MASs and an *Agent API*. Additionally the problems highlighted by Sycara can only be addressed when tools exist to help developers design, experiment with, re-build and modify MAS architectures.

Agent Oriented Engineering

Construction methods for modern software are more suited to engineers than artisans. In common with other engineering tasks, this approach takes reliable components and assembles them into larger modules, whenever possible using well established patterns of assembly. This strategy of component-assembly has offered benefits to a developing software industry as it did to a developing automotive manufacturing industry decades ago. Two key benefits relate to re-use: re-use of component parts (or component part designs) and re-use of patterns of assembly in the design phase.

One reason for the popularity/success of object orientation is that it makes this re-use easier to achieve. Component part can often be expressed as objects, easily wrapped with clearly defined interfaces. Patterns of assembly can be described in terms of stereotypical collaborations between objects (examples include model-viewer-controller architectures and collection-iterator collaborations).

In addition to re-use benefits, the engineering approach allows designer to *think differently* about the design process, use different methodologies and as a result specify systems using different architectures than would have occurred without this viewpoint. Different design methodologies and design representations often lead to different styles of solution. While two representations may be equivalent in the sense that one can be mapped to another with no loss of data (and vice versa) it is generally recognised that different representations also lend themselves to different reasoning processes both cognitively as well as mechanically (the lack of progress made by the Roman Empire in some areas of mathematics has been blamed on restrictions imposed by their number system for example (Pullen & others)). One of the claims made about the use of OO languages and design paradigms like UML is that software engineers can use representations that operate at a level of abstraction further away from the physical computing machine and that this offers them the ability to design and build larger scale, more complex systems.

"Agent-oriented software engineering introduces a new level, called the agent level, to allow the software architect modelling a system in terms of interacting agents" (Bergenti 2000). This *agent level* offers greater abstraction than the object level but can build on and complement it. The agent level has similarities with the object level: it considers discrete entities interacting through message passing, however agents tend to offer larger scale functionality than objects, responding to task-level service requests and often communicating at a level where messages request agents to perform whole tasks or sub-tasks. In some cases agents can be considered as experts albeit operating in highly restricted domains.

Re-use

Agent-oriented technology offers re-use in the form of re-useable agents and also in the form of stereotypical patterns of inter-agent communication (though it should be noted that agent use is still comparatively young so agent-level design patterns are far smaller in number and less well established than design patterns in OO environments). Additionally small collections of agents can be re-used when these collections are assembled in standard ways to perform specific high level tasks, these collections are described by some authors as *holons* (Bussmann 1998).

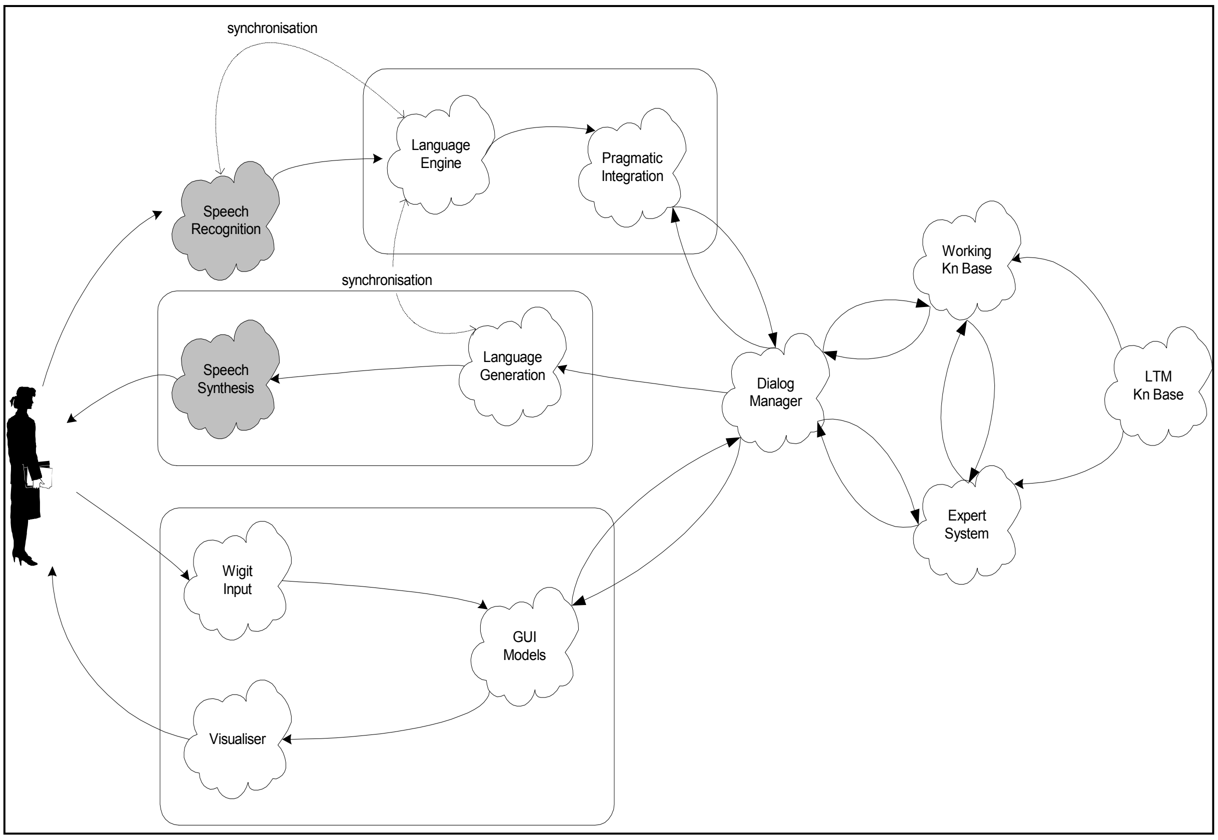


Figure 1. Example architecture for a multi-modal dialog system.

Figure 1 presents a diagram (in an informal notation) which shows the logical arrangement of agents in a MAS constructed to handle multi-modal dialog, ie: support multiple modes of user input arriving simultaneously (speech and GUI events for example) and provide multiple synchronized styles of output (speech and graphics, etc). The diagram shows a possible architecture for a *late-fusion* multi-modal dialog system. Late-fusion architectures for multi-modal dialog combine the information contained in different input modes after each has been separately analysed (conversely *early-fusion* architectures combine data from input modes such as speech and mouse click into a single utterance before considering the semantics of the user intention).

This architecture demonstrates re-use at a number of levels...

1. the MAS architecture itself is re-useable. The architecture shown in the diagram has the same merits and limitations when used as a design for a clinical support system for minor surgery (for example) as it does when used as a basis for a robot control system;
2. collaborations between collections of co-operating agents can be re-used, the dialog manager, expert system and knowledge base agents shown in the diagram form their own sub-architecture which can be used in various MAS builds;
3. individual agents can be re-used in different applications, as they are, with no modification. The language generation agent, for example, takes a structured recipe for producing a string of words, it can be re-used as-is, by *plugging it in* to various MASs;
4. other agents can be re-used in different applications/MASs by reconfiguring them in prescribed ways - the language engine shown in the example can be re-deployed by providing new lexicon and grammar.

Object oriented systems and their development platforms are geared not only to building OO programs but also to facilitate the various styles of re-use offered by object paradigms. Unfortunately agent technology is far less mature than object technology. Established development platforms do not yet exist and, although various proposals have been put forward, there is no accepted design methodology (Bergenti & Poggi 2000). This lack of appropriate tools creates a barrier for students and others who are new to MAS development. There is also a common belief that all MAS are dynamic, brokered and widely distributed over the web, with all the security and reliability issues that web activity suggests. This, coupled with the unavailability of tools, intimidates novices and puts them off investigating MAS as a suitable paradigm for their own software projects.

Design Methods For MAS

An evolving standard for a design methodology to support MAS is called *Agent UML* (AUML), it is based on the UML methodology used with object oriented systems. Bauer and others have identified UML as an appropriate basis for an agent design method (Bauer 2000 & Bauer *et al* 2000). Odell explains that new technology is best accepted by industry if it is related to existing methods or, in this case, seen to extend existing methods (Odell *et al* 2000). Users feel this minimises the learning curve associated with adopting a new technique and reduces the risk that it will suffer from premature obsolescence.

Another major reason for basing agent design on object design methods relates to similarities that exist betwen agents and objects at some levels (Iglesias 1999). However, at other levels differences between agents and objects are substantial and need new modelling constructs. Several authors have indicated UML to be insufficient for modelling agent-based systems and have suggested extensions and custominsations, e.g. (Bauer 2000; Kavi *et al* 2003; Bergenti & Poggi 2000; Yim *et al* 2000). AUML has been proposed in an attempt to standardise extensions to UML (Odell 2000). Fipa and the OMG are working to provide specifications.

The AUML community has proposed extending UML by introducing (i) Architecture Diagrams and (ii) Protocol Diagrams. We believe that, while these new diagrammatic techniques are beneficial in specifying agent behavior, there are still inadequacies. Additionally we believe that the amount of additional notation that must be added to existing diagrams in order to cover these inadequacies complicate the diagrams unnecessarily and this in turn presents an additional barrier to MAS novices. We propose instead the use of two additional diagramming techniques, one which is effectively a UML collaboration diagram and the other which explicitly captures features like agent synchronization and concurrency.

Architecture Diagrams

The architecture of a MAS is one of the most useful aspects to have diagrammed and experience shows that students are quick to produce relevant architecture diagrams. However describing them as architectures can be misleading. The term *architecture* implies some rigid unchanging structure fixed in form by design. This is not the case with MAS. Links between agents in many systems are virtual, existing only in the sense that the two agents pass each other messages. In practice it is almost certain that the message passing involves all kinds of other non-agent-based software perhaps including a messaging hub and networking protocols. Links between agents may also change over time as an agent often only forms liaisons with other agents when they provide a service required and breaking liaisons when no longer required. So, while some kind of architecture diagram is useful, it is likely to imply constraints on structure which do not exist and/or be so full of detail that it makes the fundemental relationships between agents unclear.

The work in this paper results from a primary interest in closed MASs which means that some of the difficulties associated with defining architectures for highly changeable, brokered systems are less prominent. Recognising this and also acknowledging the benefits offered by an easy to interpret system level diagram we recommend that architecture diagrams are produced but that they remain scant on details about message passing between agents, particularly when that message passing changes over time or in situations when actual identities of agents is not known until run-time (because they are brokered for example). Some authors (Bergenti & Poggi 2000 for example) recommend that architecture diagrams contain message structure details and agent class relationships (agents typically associate with classes because they are usually implemented as objects in an OO environment). We recommend that this kind of detail is intentionally omitted from architecture diagrams but is included instead in a version of a UML collaboration diagram described below. In this case architecture diagrams present generalised high-level views of MASs showing the main agent-agent relationships but remain uncluttered with implementation details.

Collaboration Diagrams & Scenarios

Collaboration diagrams are similar to architecture diagrams in that they show arrangements of agents which are linked by message passing. If a collaboration diagram represented a modular system it would show how modules plugged together but as already noted, in a MAS these links are virtual and changing. Unlike architecture diagrams, collaboration diagrams can be used to show transitional arrangements. That is: arrangements which only exist during a fixed scenario of activity or for a brief time period. Collaboration diagrams should be underpinned by descriptions of scenarios. We recommend that scenarios are used in the same way that that they are in UML and if necessary more than one level of description is written for a single scenario, the levels each describing a scenario at a different abstraction. For example in a multi-modal dialog system where the language processing agent (responsible for deriving meaning from a string of words) synchronises its language usage model with the speech input device (responsible for deriving a string of words from a spoken utterance) the language processing agent sends the speech input agent new language models whenever the programmer edits its grammar and/or lexicon. Different scenario descriptions would describe this operation with different levels of detail using an informal but precise textual style.

abstract scenario

the programmer submits a new grammar and/or lexicon to the language processor (LP) and the LP updates the language model of the speech input agent (SP).

detailed scenario

1. programmer submits a new grammar and/or lexicon to the LP via a LPuser-interface agent;
2. LP recompiles grammar/lexicon checking for errors;
3. LP analyses compiled grammar/lexicon deriving SP language model;
4. LP passes new language model to SP;
5. SP loads new language model;

   LP passes reports success to programmer via LPuser-interface.

Figure 2 shows a collaboration diagram for this scenario. The example scenario is made a little more complex if we include the detail that the LP also sends a word-use model to a language generator agent whose main role is to take a structured representation of a sentence and, from it, produce a suitable string of words for a speech output agent. A second scenario describes how speech input is forwarded to the LP which analyses it then sends a semantic representation to a dialog manager which responds by updating the user context for the LP and sending a response to the language generator. If the language generator activity is included in the first scenario and the second scenario is added to the collaboration diagram the result is as shown in Figure 3.

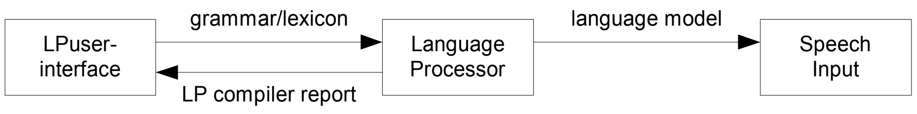


Figure 2. Collaboration diagram for simple scenario.

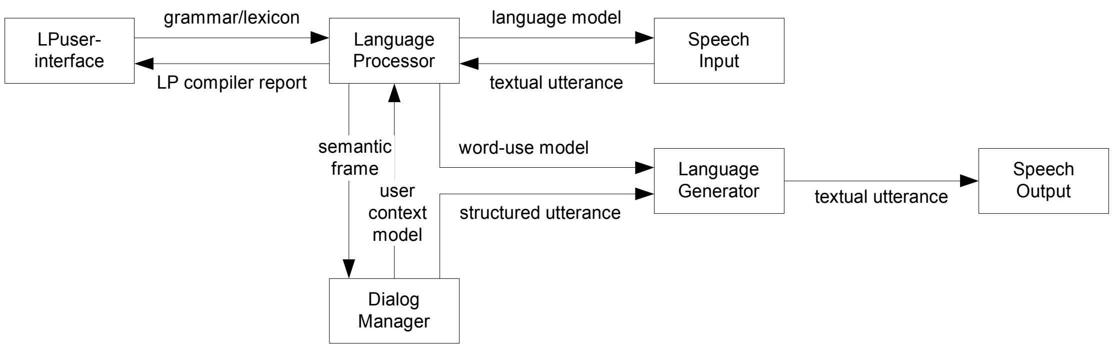


Figure 3. Collaboration diagram for more complex scenario.

Important design details relating to the sequencing and scheduling of message passing between agents are not shown in the collaboration diagram (this issue is investigated in the next section) and the diagram is approaching a saturation limit of detail beyond which it tends to muddle more than simplify a systems explanation. None the less the diagram as it is shown in Figure 3 provides useful information and can be used in the following ways...

1 to complement the architecture diagram (and check its correctness);

2 to identify closed collections of agents which may be considered to be *holons*. This can be beneficial for two reasons: (i) it identifies the possibility that a small collection of agents may be a single reusable entity and (ii) it suggests that it may be sensible to deploy the collection on a single machine (or even a single *Virtual Machine* in the case of Java);

3 to identify the types of message an agent must respond to ;

4 to identify dependencies between agents which must necessary communicate complex data to each other (an example of this from Figure 3 is the user context model).

Sequencing and Scheduling

One way in which agents differ from objects is in their run-time nature. Agents implicitly exist in their own process or thread. This means that any agent is capable of running in parallel to any other agent and also independently of any other agent. Normally this independent-parallel operation is not what is desired from a MAS. Pairs or small clusters of agents are necessarily dependent on each other, coordinating their activities in order to perform system level tasks. "*[A]gents do not only act in isolation but in cooperation or coordination with other agents. Multiagent systems are social communities of interdependent members that act individually*" (Huget 2002).

Typical examples of coordination include situations where...

1 one agent is dependent on another to provide it with data;

2 two agents must act in synchronisation to achieve an overall outcome (an example of this could be where animation is supported by a spoken commentary, the graphics animation agent and the speech output agent need to remain synchronised).

UML sequence diagrams depict the run time *life* of an object, showing its duration on *swim lanes*. When one object activates a method on another this is shown in the sequence diagram by arrows spanning swim lanes (see Figure 4 for skeleton example). The sequence diagrams assume all objects exist in a single thread so there is no ambiguity in the meaning of the diagram. An arrow from one swim lane to another implies not only that a message is passed from one object to another but also that run-time processing has shifted from one object to another. Using the same notation for agents is ambiguous since it does make it clear whether two agents should run independently in concurrent threads or whether one agent (the caller) is in a waiting state until the second agent completes its task.

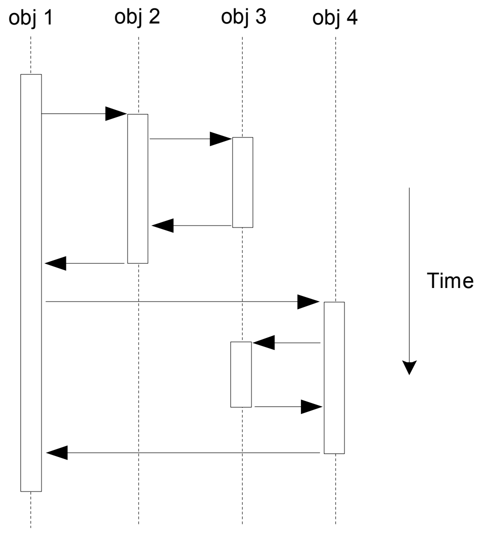


Figure 4. UML sequence diagram

The *single thread assumption* imposes another constraint on the semantics of UML, one object cannot simultaneously activate multiple other objects. This restriction should not be applied when dealing with agents. In a variety of circumstances agents may chose to do one or more of the following

1 broadcast a message to all other agents (or a subset of other agents);

2 trigger specific multiple agents;

3 initiate a schedule of activity involving many other agents.

UML sequence diagrams are also weak in expressing exceptions and exception handling through mechanisms like *try/catch* or *throw/catch*. Exceptions can occur with agents for various reasons, for example...

1 an agent fails to complete a requested task and reports some error;

2 an agent dies or fails to respond (in networked systems this could occur because the portion of the network containing the agent becomes unreachable);

3 the agent named as the recipient of a message is not registered with the system (ie: it does not currently exist);

4 the agent named as the recipient is currently busy servicing another request. While this situation is not exceptional (in the sense that the term is used here) cases of agent-busy can be conveniently handled by try/catch and allow MAS builders to specify behaviours of agents in these circumstances. It is assumed for the sake of this discussion that messages would normally be queued so if an agent is busy when it receives a new message that message would normally be entered in its message queue and dealt with some time later. Note that, as with other multiple process environments, deadlock is possible in MASs. This is more likely to occur with more complex systems which exhibit greater inter-agent dependencies.

Protocol Diagrams

The evolving proposal for Agent UML (AUML) includes agent protocol diagrams, diagrams with similarities to UML sequence diagrams (FIPA; Bergenti 2000; Bauer *et al* 2000). These diagrams intend to capture information concerned with the way agents activate each other over time and recently it has been suggested that their syntax could be extended to show sequencing and scheduling details. Some authors depict protocol diagrams, as with UML, with explicit time/swim lanes, others without them. Either way protocol diagrams explicitly represent activity with respect to time. Protocol diagrams go beyond object UML's sequence diagrams and include some important extra features. For example they can be used to depict parallel activity allowing one agent to call one or more other agents using logical connectives like AND, OR and XOR (Figure 5 shows an example of the XOR connective) and nested protocols.

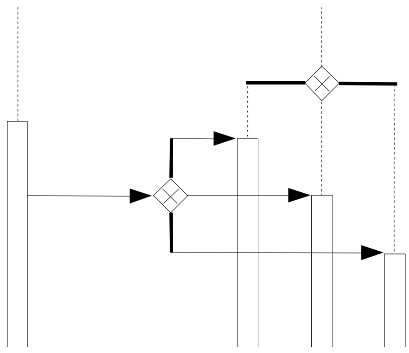


Figure 5. AUML protocol with XOR

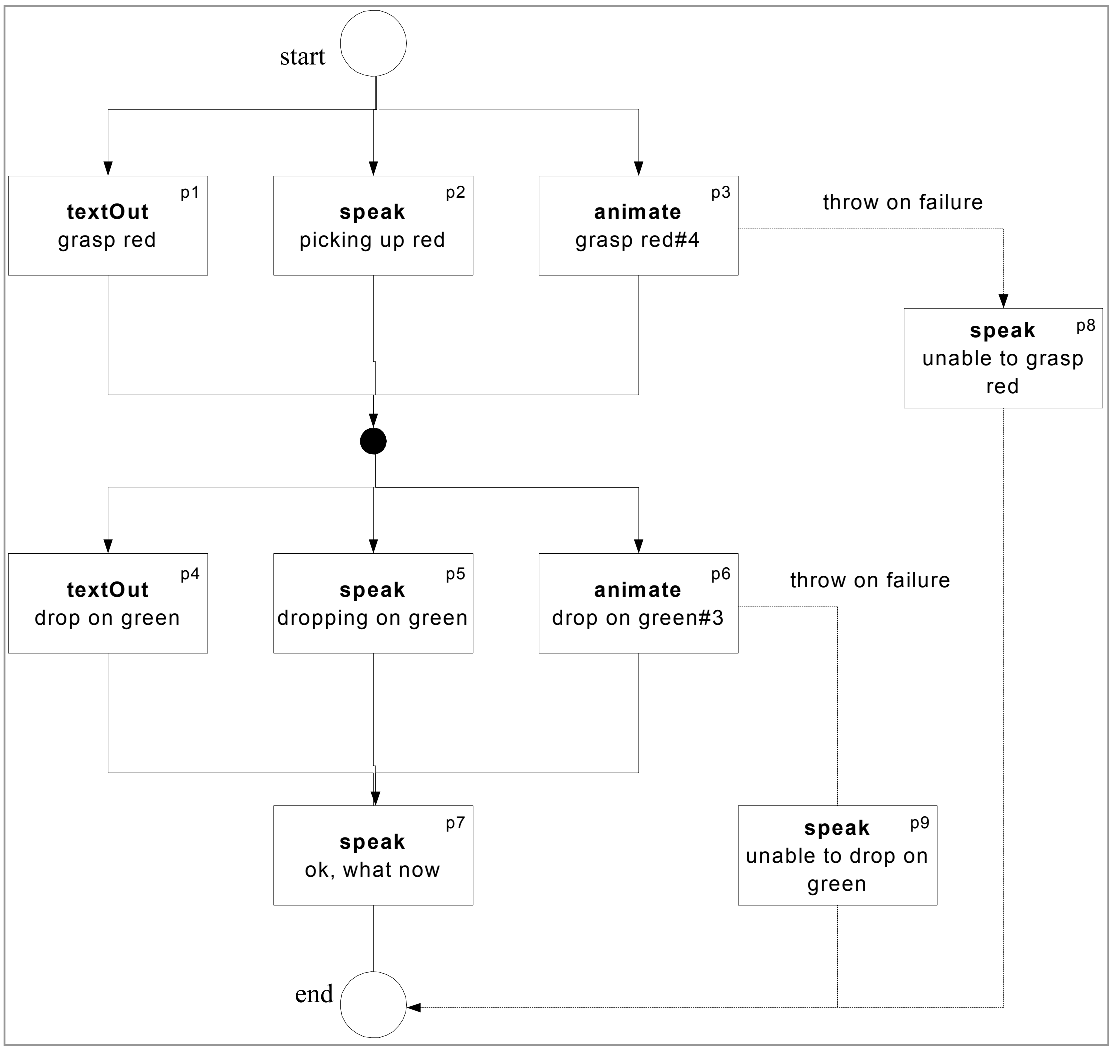
Protocol diagrams allow conditions to be associated with messages and specifications stating that messages should be sent repeatedly. Proposed extensions to current protocol diagrams (Huget 2002) include some notion of synchronisation and also exception handling. However there are difficulties with these diagrams particularly for people meeting them for the first time...

1. details relating to agent synchronisation is are not obvious on protocol diagrams which have weak synchronisation semantics and are shown syntactically by a small mark on a message arrow. Time management (delays, etc) are also minimally depicted by notes on arrows;
2. when exceptions do occur within a MAS they can initiate complex chains of behaviors which differ significantly from normal operations. Current proposals (the authors have only found few) suggest placing small textual comments on protocol diagrams to indicate exceptions (Bergent & Poggi 2000);
3. assuming that MAS builders are using a platform which allows them to specify *recipes* of non-trivial agent activity (comprising for example periods of concurrent activity, re-synchronisation, etc) it is important that a design methodology allows these recipes to be clearly expressed. Unfortunately these recipes are not easily derived from protocol diagrams.

Protocol diagrams are good for showing agent activity over time within a specific scenario but are considered here to be unsuited to depicting details of synchronization and exception handling and also describing complex recipes of agent activity. Other approaches designed to describe coordination between agents (Cost *et al* 1999; Barbuceanu & Fox 1995) target some of these problems but still do not produce diagrammatic notations which explicitly capture the synchronisation and parallelism associated with agents, neither do they have the semantic richness to express the detail of exception triggering and handling found in many real systems. In this paper we propose the use of a *flow of control* diagram to represent complex chains of agent activity. We call these "schedule diagrams".

Schedule Diagrams

A simple example is shown in Figure 6. The diagram shows the agent activity that may be required in



a simple processing of the request "*move the red block onto the green one*" in a toy world where a robot arm receives spoken instructions to move blocks around. The diagram shows a mix of sequential and parallel activity with agents named: *textOut*, *speak* and *animate*. The diagram also shows where the animation agent can throw failure.

Figure 6. Schedule diagram.

The way in which multiple agents are synchronized in practice is implementation dependent. Likewise exception-handling is implementation-dependent. *Boris* (Lynch & Rajendran 2003b), for example, is a system which provides a development platform and run-time kernel for MAS as well as an Agent API. The API contains complete agents, part formed agents and generic agents which can be specialised or extended. As part of its kernel it provides an agent-scheduler which manages synchronization. The scheduler uses *work-plans* to describe chains of agent activity. These work-plans contain concurrent sections and sequential sections (forcing synchronization). They may also contain *try/catch* forms for dealing with exceptions. Agents may explicitly throw exceptions or, in situations like agent-death or time-out, the kernel throws exceptions. An example of a coded work-plan equivalent to the diagram in Figure 6 is shown below.

```
(try (seq (par (animate (grasp red#4)
                        :throw exit1)
               (speak   (picking up red))
               (textOut (grasp red))
          )
          (par (animate (drop on green#3)
                        :throw exit2)
               (speak   (dropping on green))
               (textOut (drop on green))
          )
         (speak (ok, what now))
     )
     (catch exit1
        (seq (speak (unable to grasp red))))
     (catch exit2
        (seq (speak (unable to drop on green))))
)
```

Though no formal studies have been conducted, schedule diagrams appear to be well received by MAS novices. Their syntax clearly differentiates between, for example, one agent activating another and one *calling* another and waiting for a response. Resynchronization points can be included in the diagrams and delays are expressed with a clock icon annotated either with a delay time (250ms for example) or with a continuation condition. Exceptions are clearly marked on the diagrams and, if necessary, diagrams can be split into different sub-diagrams/sections.

Platforms, like Boris, which support try/catch for exception handling and provide agent scheduling offer obvious support for schedule diagrams but even without this support the diagrams are a useful design artefact.

Summary

Multi-agent systems (MAS) represent an important paradigm for software systems construction. Although MAS are often implemented using an object-oriented approach, they operate at a level of abstraction above the objects but still complement the object level. Some types of application, multi-modal dialog systems for example, are hard to build without a MAS approach but there are barriers to learners coming to MAS for the first time.

Processing in MAS is distributed across multiple agents with each agent often running its own process/thread concurrently with other agents. Systems which allow agents to join and leave at run-time inevitably have indeterminate structure and composition. Agents are designed to cooperate and as a result they form communities of problem solvers which can exhibit emergent behavior. All of these features make them difficult to design and document.

While there are some similarities between agents and objects, these are not enough to use object-oriented design methods without modification. Agent UML (AUML) is an evolving set of standardised extensions to object UML which aims to provide acceptable design tools for MAS builders. This paper has highlighted limitations with AUML particularly when specifying exception handling and synchronisation. We have made a case for using collaboration diagrams with agents and have introduced a new *schedule diagram* to overcome AUML's current limitations and complement the use of protocol diagrams. We believe that these additional diagramming techniques are useful tools for experienced MAS developers and, in particular, reduce the barriers faced by students and others approaching MAS development for the first time.

References

Barker, D.J, Lynch, S., Simpson, D.S. and Corbett, W.A. 1999. Speech driven natural language understanding for hands busy recording of clinical information. *Proceedings of the Joint European Conference on Artificial Intelligence in Medicine and Medical Decision Making*, Springer's Lecture Notes of Computer Science: Artificial Intelligence in Medicine, 1999.

Barbuceanu, M., Fox, M.S. 1995. COOL: a language for describing coordination in multi agent systems. *Proceedings of the First International Conference of Multi-Agent Systems* (ICMAS-95)

Bauer, B. 2000. Extending UML for the specification of interaction protocols. submitted to ICMAS 2000

Bauer, B.,Muller, J., and J. Odell. 2000 An extension of UML by protocols for multiagent interaction. In *International Conference on MultiAgent Systems* (ICMAS'00), pages 207-214, Boston, Massachussetts, july, 10-12 2000

Bergenti, F., Poggi, A. 2000. Exploiting UML in the design of multi-agent systems. *ECOOP Workshop on Engineering Societies in the Agents World* (ESAW00).

Bernsen, N. O. and Dybkjær, L. 1998. Is speech the right thing for your application? *Proceedings of the International Conference for Spoken Language Processing*, ICSLP'98, Sydney. Sydney: Australian Speech Science and Technology Association, 3209-3212.

Bekker, M.M., Nes, F.L. and Joula, J.F. 1995. A comparison of mouse and speech input control of a text-annotation system. *Behaviour and Information Technology*, 14, 1, 14-22.

Bolt, R.A. 1980. Put that there: voice and gesture at the graphics interface. *ACM Computer Graphics* 14, 3, 262-270.

Bussmann, S. 1998. An agent-oriented architecture for holonic manufacturing control. presented at First Open Workshop, IMS Europe, Lausanne, CH, 1998

Cost, R.S., Chen, Y., Finin, T., Lanrou, Y., Peng, Y. 1999. Modelling agent conversations with colored petri nets. Workshop on Specifying and Implementing Conversation Policies, Autonomous Agents, Seattle.

Damper, R.I. and Wood, S.D. 1995. Speech versus keying in command and control applications. *International Journal of Human-Computer Studies*, 42, 289-305.

Djenidi, H., Tadj, C., Ramdane-Cherif, A. and Levy, N. 2002a. Dynamic based multi-agent architecture for multimedia multimodal dialogs. *Proceedings of IEEE Workshop on Knowledge Media Networking* (KMN'02) Kyoto, JAPAN.

Djenidi, H., Ramdane-Cherif, A., Tadj, C. and Levy, N. 2002b. Generic multi-agent architectures for multimedia multimodal dialogs. In: Daniel Moldt (Ed.): *Proc. of the Second International Workshop on Modelling of Objects, Components, and Agents* (MOCA'02), P 29-46.

Gasser, L. Perspectives on organizations in multi-agent systems. In Luck, M., Marik, V., Stepankova, O. and Trappl, R. (Eds): *LNAI 2086, Multi-Agent Systems and Applications* 1-16, Springer. 2001.

Huget, M. 2002. *Agent UML Class Diagrams Revisited*. Liverpool, University of Liverpool.

Iglesias, C., Garijo, M., Gonzales, J. C. 1999. A survey of agent-oriented methodologies. *Intelligent Agents V: Proceedings of the ATAL'98*,, Springer 1999.

Lynch, S. Rajendran, K. 2003a. Modelling GUI interaction events for Java learners. Workshop on Java in Education at ACS/IEEE International Conference on Computer Systems and Applications, Tunisia 2003.

Lynch, S. Rajendran, K. 2003b. Boris - A framework for constructing multi-agent systems in Lisp and Java. *International Lisp Conference* 2003 (ILC 2003).

Kavi, K., Jung, D., Bhambhani, H. 2003. Extending UML for modeling and design of multi-agent systems. *ICSE '03 Workshop on Software Engineering for Large Multi-agent Systems* (SELMAS '03), Portland, Oregon.

Odell, J., Parunak, H. Van Dyke, Bauer, B. 2000. Extending UML for agents. *Proceedings of the Agent-Oriented Information Systems Workshop* (AOIS) at the 17th National Conference on Artificial Intelligence(AAAI),, Austin, Texas (National conf), ICue Publishing.

Oviatt, S., DeAngeli, A. and Kuhn, K. 1997. Integration and synchronization of input modes during multimodal human-computer interaction. *Proceedings of CHI'97 Conference on Human Factors in Computing Systems*, ACM, New York, 415-422.

Petitpierre, C. 2002. A design pattern for interactive applications. Ecole Polytechnique Federale de Lausanne.

Potamianos, A., Kuo, H., Lee, C., Pargellis, A., Saad, A. and Zhou, Q. 1999. Design principles and tools for multimodal dialog systems. *Proceedings of ESCA Workshop on Interactive Dialogue in Multi-Modal Systems* (Germany).

Pullen, D. http://garnet.acns.fsu.edu/~cas9574/Mesmath.doc

Resnick, M. Turtles, *Termites and Traffic Jams*., MIT Press, 1994.

Sycara, K. 2001. Multi-agent infrastructure, agent discovery, middle agents for web services and interoperation. In Luck, M., Marik, V., Stepankova, O. and Trappl, R. (Eds): *LNAI 2086, Multi-Agent Systems and Applications* 17-49, Springer.

Vidal, J., Buhler, P. 2002. Teaching multiagent systems using RoboCup and Biter, *The Interactive Multimedia Electronic Journal of Computer-Enhanced Learning*.

Yim, H., Cho, K., Park, S. 2000. Architecture-centric object-oriented design method for multi-agent systems. *4th International Conference on Multi-Agent Systems* (ICMAS).