The Zeus Agent Building Toolkit

ZEUS Technical Manual

Jaron Collis, jaron@info.bt.co.uk

Divine Ndumu, ndumudt@info.bt.co.uk

Intelligent Systems Research Group, BT Labs

Release 1.0, September 1999

Index

1 INTRODUCTION

The notion of heterogeneous autonomous agents collaborating to solve problems is a powerful metaphor for the engineering of distributed and interoperable software systems. This agent-based approach introduces a new level of abstraction — of knowledge level co-operation between autonomous systems — that enhances distributed systems interoperability, scalability and re-configurability. However, thus far, the promise of the agent approach has been largely unrealised in the distributed software engineering community. This is due to a number of factors (including the current lack of standards for agent technology), but primarily because of the inherent complexity of constructing collaborative agent systems.

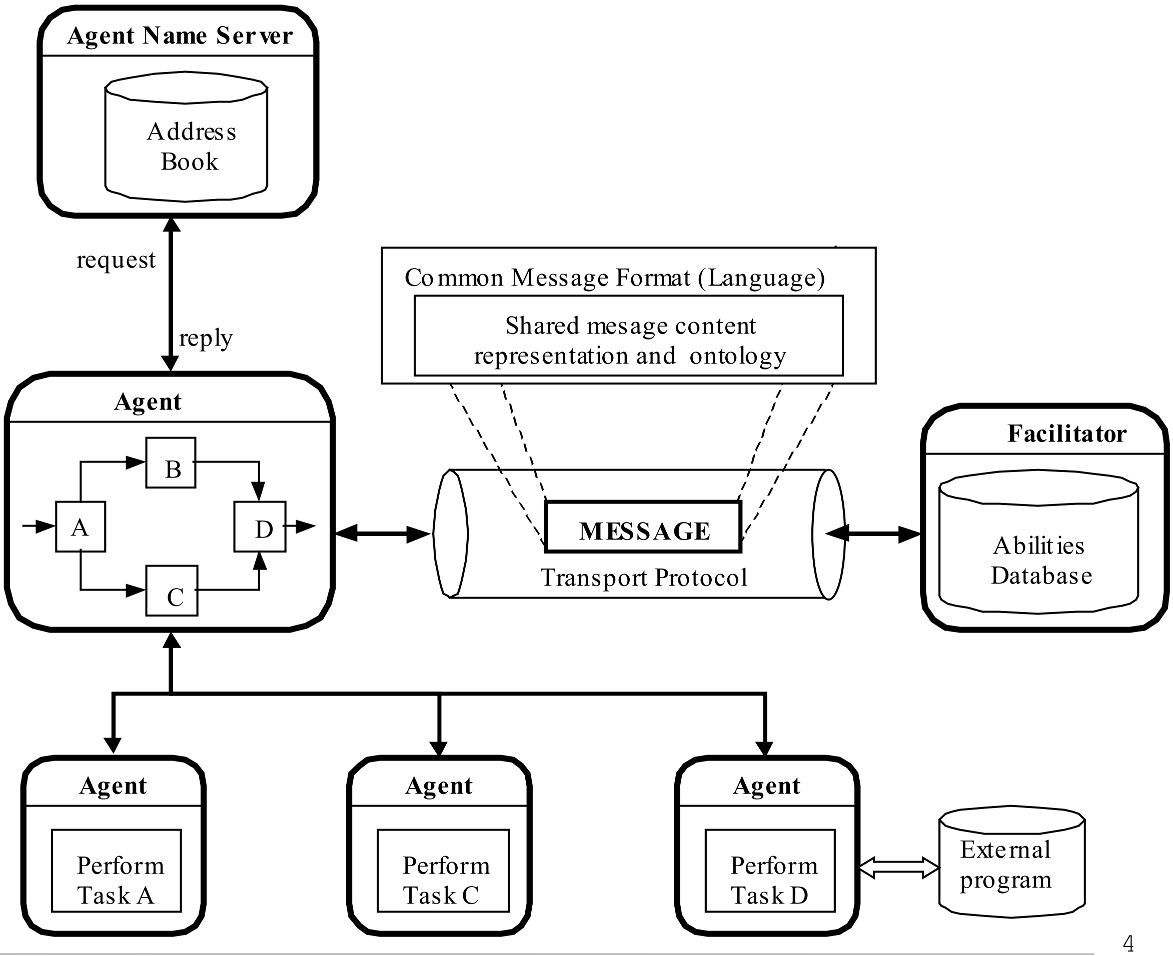
To facilitate large-scale realisation of the collaborative agent approach to distributed software engineering we felt frameworks, methodologies and toolkits were needed that would support the rapid development of multi-agent systems. This has led to the development of ZEUS, a toolkit for constructing collaborative multi-agent applications. ZEUS is a culmination of a careful synthesis of established agent technologies to provide an integrated environment for the rapid development of multi-agent systems. ZEUS defines a multi-agent system design approach and supports it with a visual environment for capturing user specification of agents that are used to generate Java source code of the agents.

1.1 ISSUES FOR BUILDERS OF COLLABORATING AGENT SYSTEMS

The need for collaboration between agents occurs for any number of reasons; however, most are rooted in the problem of scarcity of resources – computing, information, know-how, etc. Since, individual agents possess different resources and capabilities, a solution to a given problem may be beyond the capabilities of any one agent, requiring that a number of agents pool their resources and collaborate with one another in order to solve the problem.

If such collaboration proceeds at the knowledge level, [1], it places significant demands on the agents. Not least are the need for a mechanism for information discovery through which agents discover the existence, network address, capabilities and/or roles of other agents; an agent-independent inter-agent communicating language that the agents use to communicate with one another; and an ontology that defines the application domain concepts being communicated between the agents. Furthermore, for effective and coherent problem solving, the agents need mechanisms for reasoning about their own and other agents' problem solving capabilities and for co-ordinating their activities. In very dynamic environments, the problems are exacerbated by the additional requirements for data-driven reactive behaviour that integrates with the goal-driven deliberative activities of the agents. Finally, in some application domains, agent systems may need to interface with legacy systems such as databases.

Most of the issues associated with knowledge level multi-agent systems interoperation have received significant treatment, with a number of reasonably mature solutions or approaches proposed. In the following paragraphs we review the main techniques proposed for addressing the *information discovery, communication, ontology, co-ordination* and *legacy software problems*. Figure **1.1** is a context diagram illustrating the interplay between the various issues and their associated solutions.


Agent Name Server
Address Book
request
reply
Common Message Format (Language)
Shared mesage content representation and ontology
Agent
B
A
D
C
MESSAGE
Transport Protocol
Facilitator
Abilities Database
Agent
Perform Task A
Agent
Perform Task C
Agent
Perform Task D
External program

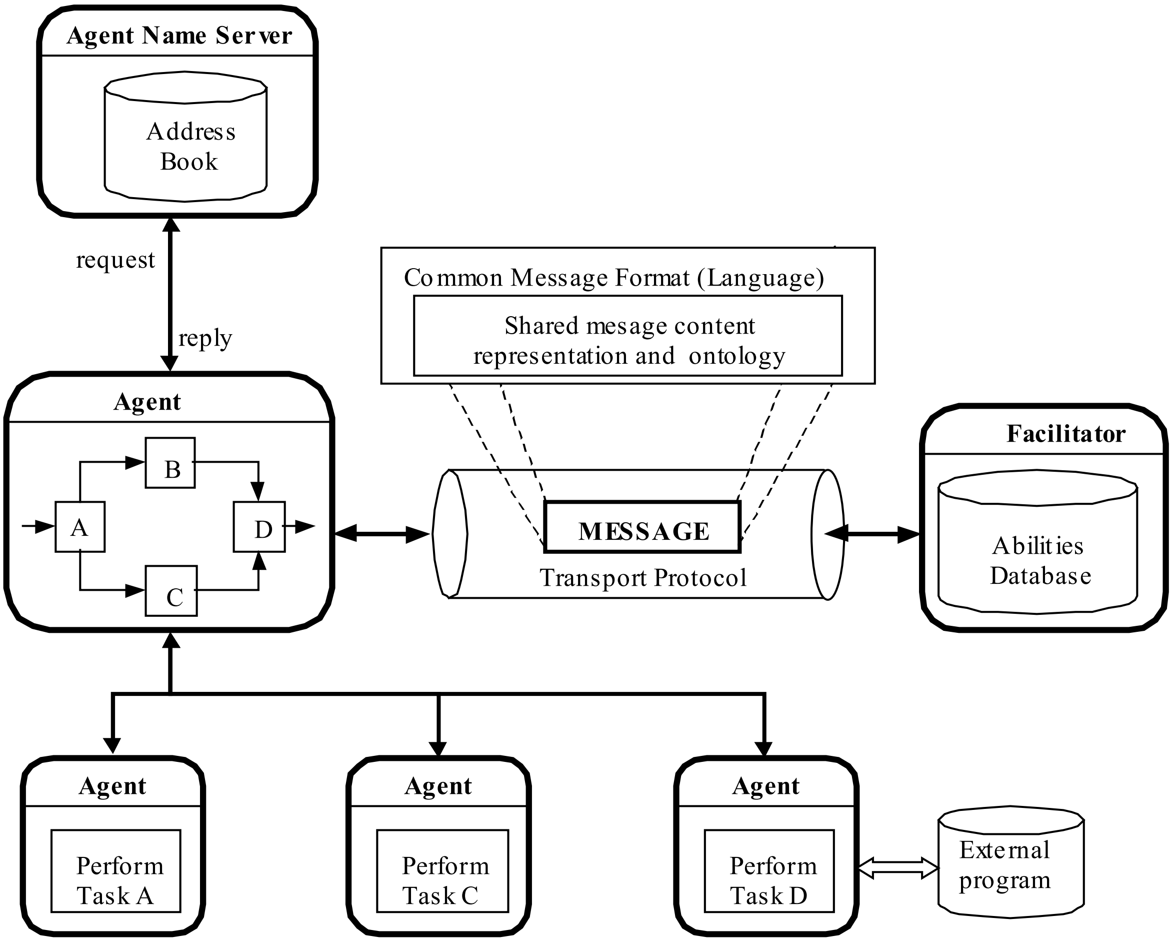


Figure 1.1: Context diagram illustrating some of the issues involved in knowledge level multi-agent collaboration. The central agent needs to perform a complex task that requires it to collaborate with other agents. To do so, it uses the Facilitator to discover the agents with the required abilities, and the Agent Name Server to determine the addresses of these agents. The inter-agent communication language is used to communicate with the Agent Name Server, Facilitator and other agents. The communication requires a shared representation and understanding of common domain concepts, i.e. a common ontology.

The Information Discovery Issue

This is typically handled using special-purpose utility agents such as *nameservers* and *facilitators* that function as society-wide white pages (address books) and yellow pages, providing a look-up service for agents' addresses and abilities respectively. Thus, agents only need to register their address with a nameserver and their abilities with a facilitator to become visible to the society. For scalability and robustness, these utility agents might be arranged in hierarchies similar to that of Internet domain nameservers.

ZEUS provides implementations of utility agents

More on this topic in **Section 3.3**

The Communication Issue

The need for an agent-independent agent communication language (ACL) has led to the development of the

Knowledge Query and Manipulation Language (KQML) [2] and FIPA ACL [3]. Most ACLs do not specify a syntax or semantics of the contents of the messages, with the rationale being that different application domains may require different content languages. Nonetheless, a number of general-purpose content languages have been developed, e.g. KIF (Knowledge Interchange Format) [4], typically used with KQML, and FIPA SL [3] the preferred content language for use with the FIPA ACL.

ZEUS supports the FIPA ACL

More on this topic in
Section 4.1

The Ontology Issue

Agents that communicate in a common language will still be *unable* to understand one another if they use different vocabularies for representing shared domain concepts. Therefore, they also need to use the same ontology or vocabularies of common concepts. This can be achieved either through general-purpose ontologies or by creating domain-specific ontologies and using inter-ontology translators to map between them. The argument for the latter is that most general-purpose ontologies are unlikely to include the intricacies of all possible domains, and are likely to be large and unnecessarily complex for most applications.

ZEUS provides tools to create new ontologies

This is described in the
ZEUS Methodology Guide

The Co-ordination Issue

Co-ordinating the behaviour of multi-agent systems is an active area of research with many techniques in use. The main approaches can be broadly classified as organisational structuring, contracting, multi-agent planning, and negotiation. In organisational structuring the prior defined structure of the society (that is, the roles of the different agents and their relationships with one another) is exploited for co-ordination, as typified by client-server systems.

Contracting as a co-ordination mechanism is typified by the classic contract-net protocol, [5], where a manager agent announces a contract, receives bids from other interested contractor agents, evaluates the bids and awards the contract to a winning contractor. Some interesting alternatives/variants to the contract-net protocol include various auction protocols such as the *english, dutch* and *double* auctions.

In multi-agent planning, the agents utilise classical AI planning techniques to plan their activities, resolving any foreseen conflicts. The planning normally takes one of two forms, centralised planning – in which a central agent performs the planning on behalf of the society, or decentralised planning – in which the agents exchange partial subplans, progressively elaborating the overall plan and resolving conflicts in it.

With negotiation, the agents engage in dialogue, exchanging proposals with each other, evaluating other agents' proposals and then modifying their own proposals until a state is reached when all agents are satisfied with the set of proposals. Typical negotiation mechanisms are based on game theory, on some form of planning, or on human-inspired negotiations.

ZEUS supports all these forms of agent co-ordination

More on this topic in
Section 4.2

Integration with Legacy Software

As agents are not intended as replacements for legacy software, they must be able to interact with it. Generally speaking there are three possible approaches: the software could be rewritten, but this is a costly approach. Alternatively a separate piece of software called a *transducer* could be employed to act as an interpreter between the agent communication language and the native protocol of the legacy system. Or thirdly, the *wrapper* technique could be used to augment the legacy program with code that enables it to communicate using the inter-agent language.

ZEUS agents can act as wrappers around legacy systems

More on this topic in
Section 4.5

Throughout this document we shall return to these issues, and explain how they are resolved within the ZEUS toolkit.

In Section 2, we outline the philosophy and assumptions underpinning the design of our agent building toolkit. Furthermore, we specify the type of agent systems the toolkit is designed to create, as well as the class of application domains of these agents.

In Section 3, we describe the ZEUS toolkit architecture; this section provides an overview of the design of the entire toolkit. We describe its Agent Component Library, which is used to implement the generic ZEUS agent, the visual agent creation environment, and the suite of ZEUS utility agents.

Section 4 describes the implementation details of the Agent Component Library. We concentrate on some of the main components of a typical ZEUS agent, for example, the communication manager, the co-ordination engine, the planner, the internal event model, and the mechanisms for connectivity to external (legacy) systems.

2 THE ZEUS DESIGN PHILOSOPHY

The aim of the ZEUS project was to facilitate the rapid development of new multi-agent applications by abstracting into a toolkit the common principles and components underlying some existing multi-agent systems. The idea was to create a relatively general purpose and customisable, collaborative agent building toolkit that could be used by software engineers with only basic competence in agent technology to create functional multi-agent systems. Thus, our design philosophy was to encapsulate the following principles:

- Firstly, the toolkit should clearly delineate between *domain-level* problem solving and *agent-level* functionality. The latter covers the application-independent multi-agent issues such as communication, co-ordination, task execution and monitoring, exception handling, etc. while the former covers the acquisition, representation and use of domain-specific knowledge in problem solving. The intention being with the agent-level functionality provided, the developers could concentrate on implementing the domain-specific problem solving abilities of their agents.
- Secondly, use of the toolkit should be based on the 'visual programming' paradigm. Hence the toolkit would support the agent creation process by providing structured menus and tables that would enable application developers to configure the functionality and modalities required of their agents as simply as possible.
- Thirdly, the toolkit should support an open design to ensure it is easily extensible. Thus, expert users should be able to easily add to the library of agent level components, and configure new agents using a combination of user-defined and system-supplied components.
- Fourthly, we intended to utilise ‘standardised’ technology wherever feasible, or be designed with standardisation in mind. We felt that standardisation was essential was for the industrial uptake of agent technology. This way, we envisaged components which could later be replaced with little difficulty by ‘more standardised’ components. This is typified by our adoption of the most 'standardised' agent communication language currently available.

ZEUS Functional Requirements

Having briefly described our design philosophy, we now consider the requirements from the viewpoint of a user of the toolkit. Viewed from a user-centred perspective, it was required that the toolkit allow users to

- *configure* a number of different agents of varying functionality and behaviour;
- *organise* the agents in whatever manner using system-supplied organisational relationships;
- *imbue* each agent with selected system-supplied and/or user-defined communicative and co-ordination mechanisms;
- *supply* each agent with the appropriate application-specific problem solving code; and
- *generate* automatically the executables for the agents.

In addition, we also mandated the following of the toolkit:

- it should provide predefined information discovery agents such as nameserver and facilitator agents
- it should also provide extensive facilities for visualising and debugging societies of ZEUS agents.

ZEUS Agent Assumptions

It is important to summarise at this point some fundamental assumptions made about the type of agents whose creation the toolkit is designed to facilitate, and also to describe the typical application domains of these agents. The principal assumptions made regarding the agent behaviour are that the agents are:

- deliberative, goal-directed and rational;
- always truthful when dealing with other agents;
- versatile, i.e. can have many goals and can engage in a variety of tasks; and

- temporally continuous.

The agents should be deliberative in the sense that they should explicitly reason about their actions in terms of what goals to pursue, when to adopt new goals and when to abandon existing goals. In addition, the requirement for goal-directed behaviour implies the agents only select actions that they expect in some way to advance the attainment of their desired goals. Furthermore, they only abandon goals when certain either that they cannot achieve the goals or that the motivations for achieving the goals no longer hold. The rationality assumption implies the agents adopt only actions that they expect to maximise their expected utility. That is, given a choice of actions, an agent would select a subset that it expects can be performed given the available time and resources, and further, that should lead to the maximum possible benefits.

Typical Application Domains for ZEUS Agents

The typical application areas of the agents were expected to be task-oriented domains such as service provisioning, resource/process management, and supply chain management. A number of characteristics of these domains are important:

- firstly, given a goal, an agent creates a plan of action to achieve the goal, and such plans require explicit reasoning about the preconditions and effects of domain actions given limited time and other resources;
- secondly, typical problem solving requires co-operation with other agents;
- thirdly, while the environment of the agent is dynamic, the rate of change of the environment is at least an order of magnitude less than the reasoning time of the agent. That is, there is a less than five percent chance that the external conditions on which a plan of action is based will change while an agent is in the process of creating the plan; and
- finally, the role of agents in such domains is typically to reason about how and when to configure, activate or deactivate external systems that perform the real work in the domain. Thus, domain problem solving is not really embedded within the agent *per se* but external to it. The agent, in effect, possesses a logical model of the external system that it uses in making control and management decisions about the system.

In the next section, we describe the ZEUS toolkit that was developed to meet the above requirements. In subsequent sections we provide an example of the use of the toolkit, and then proceed to describe the design and implementation of the main components of ZEUS agents.

Several typical applications are described in the **ZEUS Role Model Guide**

3 THE ZEUS TOOLKIT ARCHITECTURE

The ZEUS toolkit consists of a set of components, written in the Java programming language, that can be categorised into three functional groups (or libraries) as depicted in Figure 3.1: an agent component library, an agent building tool and a suite of utility agents comprising nameserver, facilitator and visualiser agents. In the following subsections, we describe in turn the ZEUS agent component library, the agent building approach and its associated environment, and the suite of utility agents.

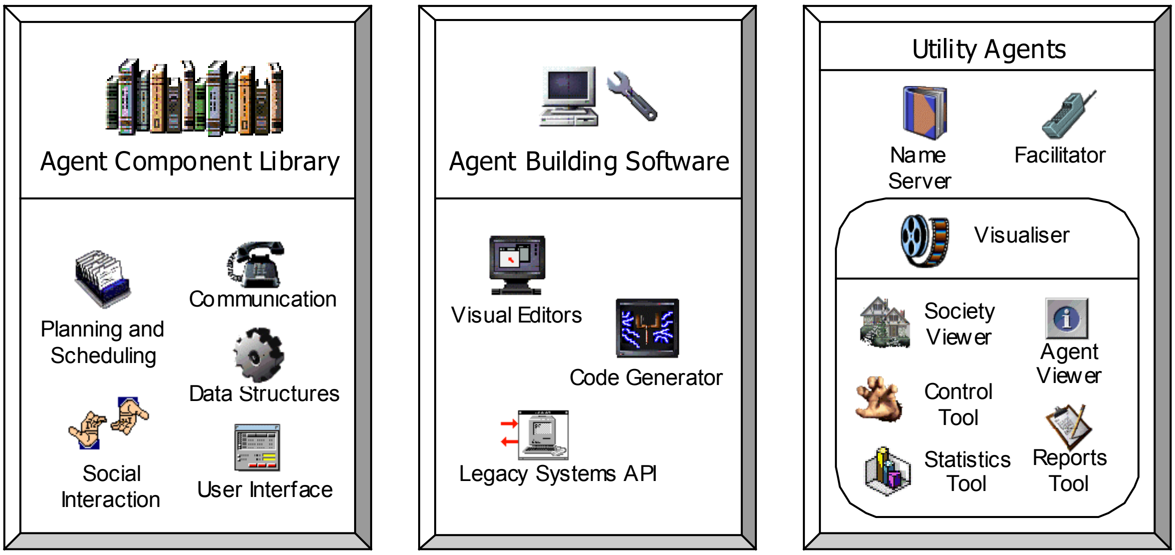


Figure 3.1: Components of the ZEUS agent building toolkit.

3.1 The Agent Component Library

The Agent Component Library is a collection of classes that form the building blocks of individual agents. Together these classes implement the application-independent *agent-level* functionality required of collaborative agents. The contents of this library address the issues identified in Section 1 including communication, ontology, co-ordination (or social interaction).

For **communication** the Agent Component Library provides:

- a performative-based agent communication language, in our case KQML;
- an asynchronous socket-based message passing system;
- an editor for describing domain-specific ontologies — the domain concepts that are defined using the ontology editor are used as part of the content language within the ACL; and
- a frame-based knowledge representation language for representing domain concepts.

Next, for **reasoning and multi-agent co-ordination**, the Agent Component Library provides:

- a general purpose planning and scheduling system suitable for typical task-oriented application domains, and the co-operative problem-solving inherent to these applications (see Section 3), and
- a co-ordination engine that controls the social behaviour of an agent, i.e. when and how it interacts with other agents and the types of contracts it sets up with them.

The functioning of the planner and co-ordination engine are influenced by the agent's knowledge context, i.e. its available resources and competencies, its organisational relationships with other agents and its available co-operation strategies. Thus, to support these two components, the Agent Component Library also provides:

- a library of predefined re-usable co-ordination protocols, e.g. contract-net and various auction protocols.
- a number of predefined organisational relationships. The current set of relationships includes *superior*, *subordinate*, *co-worker* and *peer* relations. Agents that are defined as superior to other subordinates agents can delegate tasks to their subordinates. Agents that belong to the same static 'community' can be declared as co-workers, meaning they prefer to interact with one another. The peer relationship is the default, and it does not impose any restrictions on interaction. (Remember, we noted in Section 1 that organisational structuring affects the co-ordination of a multi-agent set-up).
- knowledge representation mechanisms and databases for describing and storing the resources and competencies of an agent.

The Generic ZEUS agent

Together, the components of the Agent Component Library enable the construction of an application-independent generic ZEUS agent that can be customised for specific applications by imbuing it with problem-specific resources, competencies, information, organisational relationships and co-ordination protocols. Figure 3.2 shows the architecture of the generic ZEUS agent that is not too dissimilar from other collaborative agent architectures in the literature.

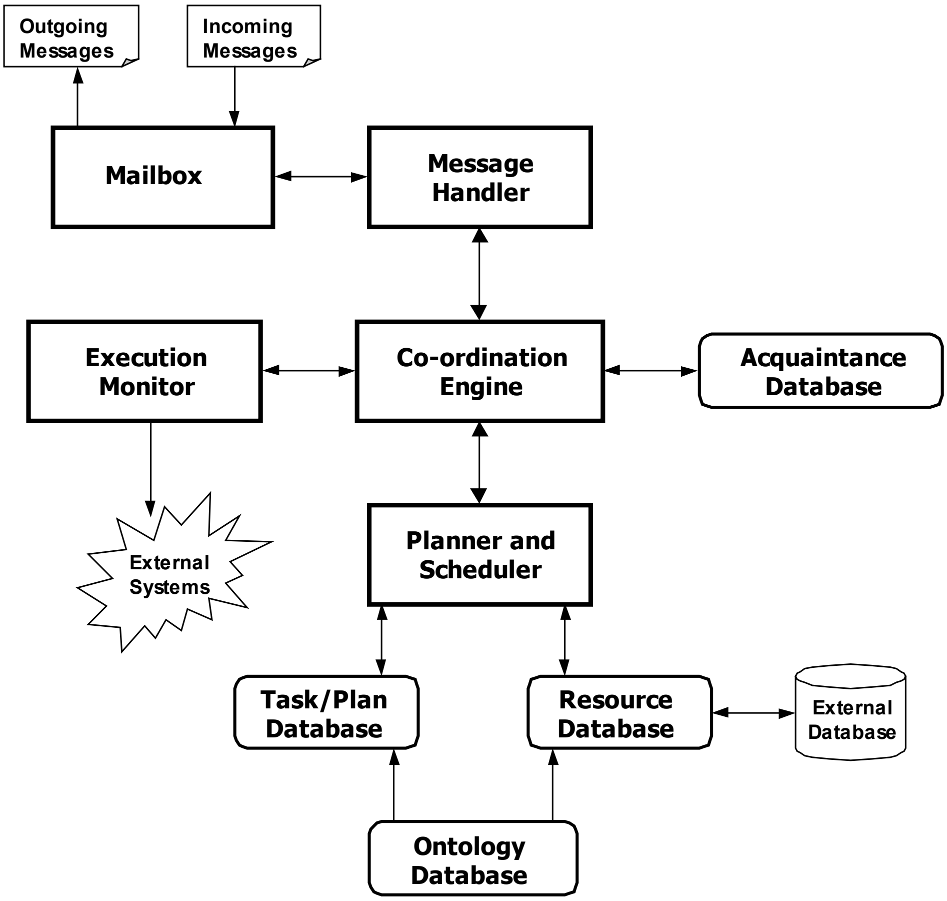


Figure 3.2: Architecture of the generic ZEUS agent

As Figure 3.2 depicts, the generic ZEUS agent includes the following components:

- a Mailbox that handles communications between the agent and other agents.
- a Message Handler that processes incoming messages from the Mailbox, dispatching them to the relevant components of the agent.
- a Co-ordination Engine that makes decisions concerning the agent's goals, e.g. how they should be pursued, when to abandon them, etc. It is also responsible for co-ordinating the agent's interactions with other agents using its known co-ordination protocols and strategies, e.g. the various auction protocols or the contract net protocol.
- an Acquaintance Database that describes the agent's relationships with other agents in the society, and its beliefs about the capabilities of those agents. The Co-ordination Engine uses information contained in this database when making collaborative arrangements with other agents.
- a Planner and Scheduler that plans the agent's tasks based on decisions taken by the Co-ordination Engine and the resources and task specifications available to the agent.
- a Resource Database that maintains a list of resources (referred to in this paper as *facts*) that are owned by and available to the agent. The Resource Database also supports a direct interface to external systems, which allows it to dynamically link to and utilise proprietary databases.
- an Ontology Database that stores the logical definition of each fact type — its legal attributes, the range of legal values for each attribute, any constraints between attribute values, and any relationships between the attributes of the fact and other facts.
- a Task/Plan Database that provides logical descriptions of planning operators (or tasks) known to the agent.
- an Execution Monitor that maintains the agent's internal clock, and starts, stops and monitors tasks that have been scheduled for execution or termination by the Planner/Scheduler. It also informs the Planner of successful and exceptional terminating conditions of the tasks it is monitoring. In order to manage tasks, the Execution Monitor also has a direct interface to external systems. It is assumed that the domain realisations of tasks are external programs.

In the next subsection, we describe a typical *use case scenario* to illustrate the flow of information and control in the generic ZEUS agent.

Information and control flow in the generic ZEUS agent

Imagine a message from another agent is received by the agent's Mailbox, which passes the message to the Message Handler for processing. On receipt of the message, the Message Handler interprets it as a request to achieve a goal. Hence, it forwards the message to the Co-ordination Engine to determine whether to achieve the goal and if so, to devise and co-ordinate an appropriate plan of action.

The Co-ordination Engine decides to attempt the goal, and invokes the Planner to construct a plan to achieve the goal. The Planner creates a plan for the goal, utilising action descriptions from its Plan Database, and reserving the resources that are required by the plan and available in its Resource Database. However, the Planner finds that there are some other resources that are required by the plan, but which are not available in its Resource Database, and which it cannot produce. Thus, it calls the Co-ordination Engine to seek external assistance in producing those resources.

The Co-ordination Engine then begins to attempt to contract out the task of providing the required resources at the required time. To do this, it checks its Acquaintance Database for the names of other agents that it believes can produce the required resources. Finding no acquaintance agents with the appropriate abilities, the Engine uses the Mailbox to send a message to a known facilitator, requesting a list of all "active" agents with the required abilities. On receipt of a reply from the facilitator, the Mailbox forwards the reply message to the Co-ordination Engine (through the Message Handler).

Now, given the list of agents with the needed abilities, the Co-ordination Engine first stores this information in its Acquaintance Database, and then proceeds to send messages to the agents, asking them to bid for a contract to produce the required resource. Again the outgoing messages are sent through the Mailbox and their replies returned to the Co-ordination Engine via the Mailbox and Message Handler.

Once all contractor agents have returned their bids for the tasks, or the reply deadline has expired, the Co-ordination Engine passes the returned bids to the Planner, which selects suitable contractors for providing the required resources. The suitability of each bid depends on factors such as its cost, and how well it fits in with the overall plan to achieve the original goal. With the bid selections made and the plan completed, the Planner returns to the Co-ordination Engine a list contractor agents to whom send contract award messages should be sent, and another list to whom the Engine should send bid rejection messages.

However, before sending out the contract award and bid rejection messages, the Co-ordination Engine first sends a message to the agent that originally asked it to achieve the goal, informing the agent that it can perform the goal and the cost of doing so. Next, the Engine waits for a response to its bid. If a favourable response is received, it then sends out the contract award and bid rejection messages to its own contractor agents and informs the Planner that the plan for the goal should be executed when appropriate. If, on the other hand, an unfavourable response was received, bid rejection messages are sent out to all contractor agents, and the Planner is told to cancel the plan.

Once a scheduled plan is ready for execution, the Execution Monitor executes the actions specified in the plan by invoking the external program declared in each action description. If the entire plan is successfully executed, the final results are sent through the Co-ordination Engine and Mailbox to the agent that requested the goal.

As can be seen from the use case scenario, the components of the Agent Component Library work together to provide the necessary agent-level functionality. For instance, the Mailbox and the Ontology Database facilitate communication. The former provides agents with the ability to send and receive messages in a ‘standard’ format, whilst the latter enables each agent to understand what other agents communicate to it. Once agents can communicate, we can raise the level of abstraction to the co-ordination level (or social interaction), wherein bargaining and negotiating is possible. This is realised via the Co-ordination Engine employing various defined co-ordination protocols. It is also clear from this example that co-operative problem solving between agents in task-oriented domains requires some planning and scheduling capabilities.

In the next section, we describe the second major sub-library of the ZEUS toolkit — the agent building software.

3.2 THE ZEUS AGENT BUILDING SOFTWARE

The principle underlying the ZEUS toolkit is that application-specific agents can be constructed by configuring the generic ZEUS agent, and equipping it with the necessary application functionality. To facilitate rapid development, the ZEUS toolkit provides high-level agent development approach that hides the complexities of the Agent Component Library from the agent developer. This approach has two key aspects:

- an agent creation methodology, which guides the developer through the analysis and design of the intended system, and
- a *visual* agent development environment that supports the creation methodology

The agent creation methodology is vital to the use of the ZEUS toolkit, and is described in detail in its own document, see the **ZEUS Realisation Guide**.

This section will briefly discuss the ZEUS Agent Generator, the suite of integrated editors that support the ZEUS agent design approach. To facilitate ease of use, the editors have been designed to enable users to interactively create agents by visually specifying their attributes. The current suite of editors includes:

- An Ontology Editor for defining the ontology items in a domain. Concept categories — referred to as fact templates — can be created for application domains, with the concepts related to one another as appropriate through object-oriented style inheritance and/or composition. Fact objects are defined in terms of their attributes and the valid value ranges for each attribute. Attribute values can be primitive types, lists, other facts or constraint expressions that should ultimately resolve into a primitive type, list or fact.
- A Fact/Variable Editor for describing specific instances of facts and variables, using the templates created using the Ontology Editor.
- An Agent Definition Editor for describing agents logically. This involves specifying each agent’s

tasks, its initial resources, and the dimensions of its plan diary.

- A Task Description Editor for specifying the attributes of tasks and for graphically composing summary tasks.
- An Organisation Editor for defining the organisational relationships between agents, and agents' beliefs about the abilities of other agents.
- A Co-ordination Editor for selecting the set of co-ordination protocols with which each agent will be equipped.

Thus, in order to generate the code for a specific application, the Generator tool *inherits* code from the Agent Component library, and *integrates* it with the data from the various visual editors. The resulting programs can be compiled and executed normally.

3.3 THE ZEUS UTILITY AGENTS

It is standard practice for a distributed society of agents to have an infrastructure of utility services. The ZEUS suite of utility agents consists of a *nameserver* and a *facilitator* agent that facilitate information discovery, and a *visualiser* agent for visualising or debugging societies of ZEUS agents. A ZEUS agent society may contain any number of these utility agents, with at least one nameserver agent. All three utility agents are constructed using the basic components of the Agent Component Library, and are in fact simplifications of the generic ZEUS agent.

Nameserver agents have only a Mailbox and Message Handler, the components needed for receiving and responding to agents' requests for the addresses of other agents. In addition, nameserver agents maintain a society-wide clock; thus, on initialisation, an agent registers with a nameserver and synchronises its internal clock to that of the nameserver. However, although a society may contain multiple nameserver agents, only the very first one defines *time-zero*.

Facilitator agents have a Mailbox and Message Handler for receiving and responding to queries from agents about the abilities of other agents, and an Acquaintance Database for storing the abilities of the agents. They function by periodically querying all the agents in the society about their abilities, and storing the returned information in their Acquaintance Database. Also, individual agents might advertise their abilities to facilitators. Thus, when an agent wants to find other agents that have a particular competence, they can simply send an appropriate query message to a facilitator agent.

Visualiser agents can be used to view, analyse or debug societies of ZEUS agents. They function by querying other agents about their states and processes, and then collating and interpreting the replies to create an up-to-date model of the agents' collective behaviour. This model can be viewed from different perspectives through visualisation tools supported by the visualiser agents. The current tools include:

- a **Society Viewer** that shows all the agents in a society and their organisational inter-relationships. It can also show the messages exchanged between the agents during problem solving.
- a **Reports Tool** that shows the society-wide decomposition/distribution of active tasks and the execution states of the various tasks.
- an **Agent Viewer** that enables the internal states of agents to be observed and monitored.
- a **Control Tool** that is used to remotely review and/or modify the internal states of individual agents. Thus, an agent's behaviour can be redefined at runtime by using this tool to modify its task, resource, or organisational databases, or even by providing it with new message processing rules and/or co-ordination graphs. In this regard, the control tool is effectively an online version of the Agent Building Software. This tool also facilitates administrative management of agent societies, e.g. agents can be killed or suspended, they can be given news goals, or their old goals can be modified.
- a **Statistics Tool** that displays individual agent and society-wide statistics in a variety of formats.

The multi-perspective visualisation approach provided by the visualisation tools gives users the flexibility to choose what is visualised, how it is visualised and when it is visualised. The visualisation tools are generally used *online*, to visualise the interactions in a multi-agent society live, as they happen. However, the society, report and statistics tools can also operate *off-line* by recording agents' interaction sessions to a database. Once stored, recorded sessions can be replayed, video-recorder style, using the forward and rewind buttons.

The features of the various Visualiser windows, and how they can be used to analyse and debug agent systems, are described in the **ZEUS Runtime Guide**.

How Utility Agents Start Up

Consider a small agent society consisting of three 'task' agents and the standard three 'utility' agents. The interactions that occur between them at start-up are shown in the interaction diagram of Figure 3.3, this shows how the agents, (the vertical dashed lines), interact with each other, (shown by the horizontal arrows). All interactions are achieved through the standard ZEUS message passing mechanism.

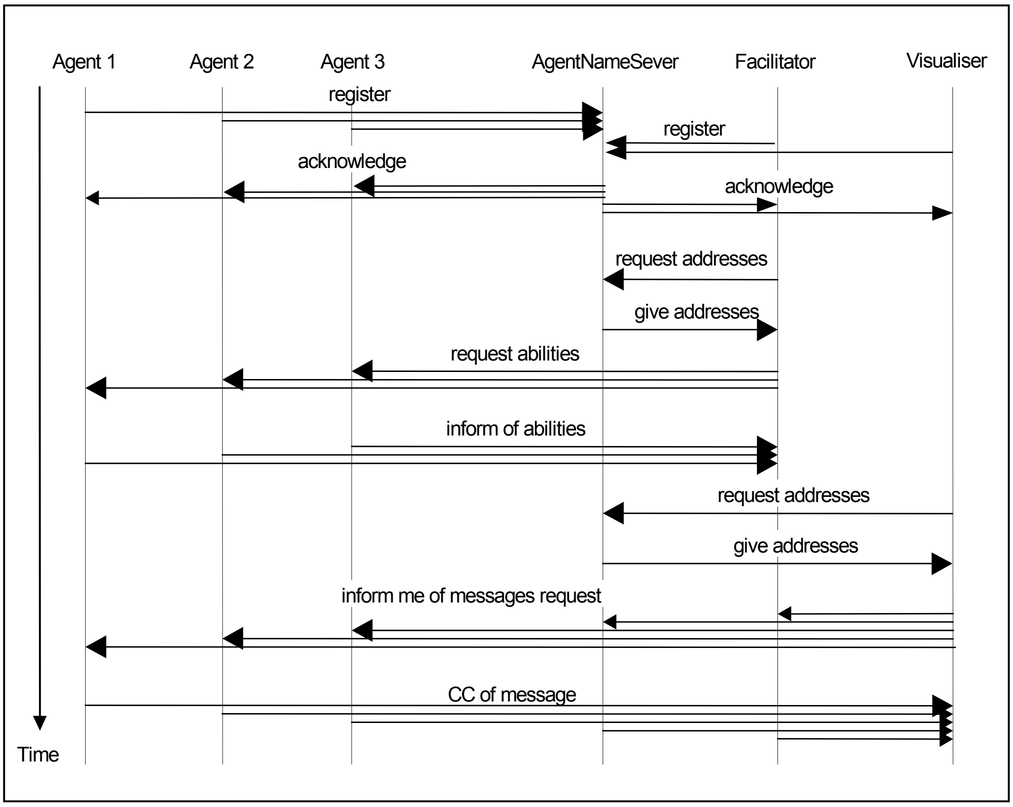


Figure 3.3: An interaction diagram of a newly created agent society

Figure 3.3 provides a useful insight into how the agents function. The interactions shown are time ordered, with those at the top occurring before those further down. The direction of the arrows is also significant, as it shows which agent initiated interaction and who responded to it. At this point it may also be informative to consider the intra-agent interactions, i.e. how the component parts of an agent work together. So next we shall consider the design and implementation of the main components of the generic ZEUS agent.

4 THE AGENT COMPONENTS

This section describes the implementation details of the Agent Component library. We concentrate on some of the main processing components of the generic ZEUS agent:

- the communication mechanism,
- the co-ordination engine,
- the planner,
- the internal event model, and
- the connection mechanisms to external (legacy) systems.

The main design principle underpinning the design of the various components was that the components should permit some form of declarative specification of message processing, co-ordination and planning behaviour. Thus, the behaviour appropriate to an agent should be specified declaratively using the Generator tool and processed accordingly at runtime by the agent. As agent behaviour is declaratively specified, it can be modified dynamically, even at runtime; as a result the generic ZEUS agent functions like an 'interpreter' of specified behaviour. In subsequent subsections, we describe how this was achieved.

4.1 THE COMMUNICATION MECHANISM

Communication between ZEUS agents is via point-to-point TCP/IP sockets, with each message communicated as a sequence of ASCII characters. This is realised through the combined actions of an agent's Mailbox and Message Handler components, permanent threads that run concurrently for as long as the agent is alive. The creation order and interactions between the threads of the communication mechanism are shown in Figure 4.1, (where permanent threads are dashed lines and transient threads are shown as dotted lines).

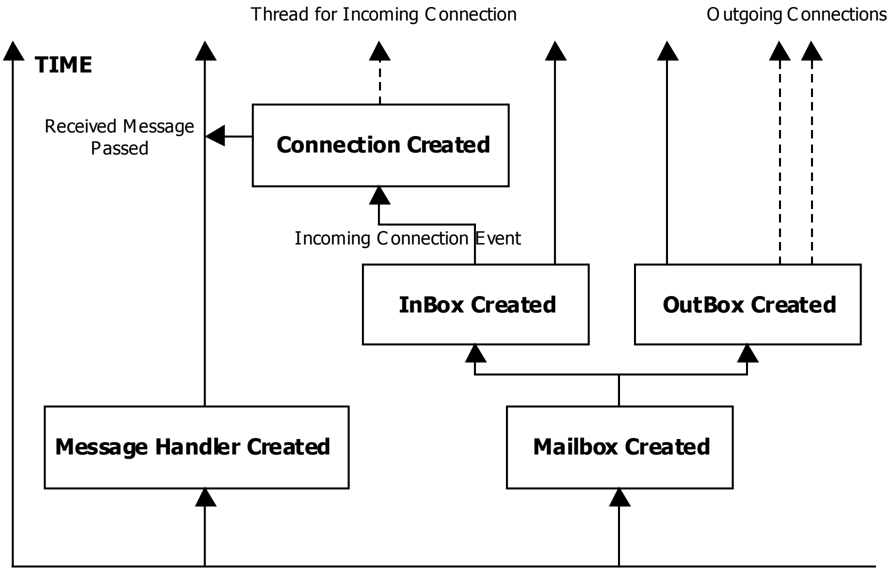


Figure 4.1: A timeline of the active threads of the Zeus Communication Mechanism

The Mailbox

The Mailbox is responsible for creating and reading TCP/IP sockets to send and receive messages. It maintains two independent threads of activity — one, a reader thread, continually listens for incoming socket connections. When one is detected it will create a new transient thread to read the message and deliver it to the Message Handler, which will process it. This approach delegates responsibility for reading messages to the new connection thread, leaving the main 'InBox' thread free to continue listening for incoming messages, (thus enabling several messages to be received simultaneously). When the incoming message is read, the connection thread terminates.

The other Mailbox thread is the 'OutBox'. When this is given a message to dispatch it creates a transient thread to open a socket to the recipient. If the connection is made the message is then streamed down the socket, and when finished the writer thread terminates. Again this approach means an agent can dispatch more than one message at a time.

The writer thread of the Mailbox continually checks a priority FIFO outgoing-message-queue for messages to dispatch. For each message awaiting dispatch, it queries the message object for the intended recipient, and looks up a local address book for the recipient's address. If the address is found, the writer opens a network socket connection to the agent at the specified address. Next, it serialises the message object as an ASCII sequence onto the network connection. If the recipient's address is not found, the writer stores the message object onto a holding buffer, and queries known NameServer agents for the required address. This embedded query utilises recursively the same Mailbox and Message Handler functionality. Once the message recipient's address is received, the writer removes the relevant message from the holding buffer and proceeds to dispatch the message. In the event that no address is found or network communications fails, a suitable error message is generated, which the writer adds to the reader's incoming-message-queue to be processed as a normal incoming message.

Why TCP/IP?

TCP/IP was deliberately chosen as the transport protocol for ZEUS messages, in preference to object-oriented middleware solutions like CORBA. As the lowest common denominator, the transport protocol used will ultimately dictate the portability of the agents. Hence our rationale was to choose the most ubiquitous standardised protocol, and for the foreseeable future that is likely to be TCP/IP. It also has the additional advantage of being lightweight, meaning we could implement functionality in the agent layer rather than having to rely on services in the transport layer: the facilitator agent provides a good example.

As all aspects of ZEUS agent communication are encapsulated inside the Mailbox, it would be perfectly possible to replace the TCP/IP mechanism with a middleware alternative, should that be felt necessary.

Messages sent through sockets are received message as a stream of ASCII characters, which are then parsed into a *Performative* object and queued onto a priority FIFO incoming-message-queue[1]. Performatives form the basis of the most inter-agent communication languages, and those used in ZEUS are explained in section 4.1.1.

[1] Most of the queues used in the ZEUS implementation were designed as blocking queues. When a blocking queue is empty, any thread trying to remove an element from the queue blocks until another thread adds an element into the queue.

4.1.1 The Language of Communication

Most agent communication languages (ACLs) are based on speech act theory [6], wherein human utterances are viewed as actions in the sense of actions performed in the everyday physical world (e.g. picking up a block). Hence, ACLs specify message types called **performatives**, such as *ask*, *tell*, or *achieve,* which by virtue of being sent from one agent to another, are assumed to effect some illocutionary actions in the receiving agent.

Obviously, inter-agent compatibility will be impossible until all parties adopt the same agent communication language, and fortunately ACL standards do exist. All ZEUS agents communicate using messages that obey the FIPA 1997 ACL specification, which is described in http://www.fipa.org/spec/f8a22.zip.

The syntax of this language is included in the ZEUS toolkit in the file **zeus/concepts/Performative.jj,** which is used by the agent's internal parser to formulate and decode messages. This syntax is used to construct instances of the Performative class, which have the following attributes:

```
Performative(
        type:           /* performative type, e.g. inform, cancel etc. */
        sender:         /* name of agent sending message */
        receiver:       /* name of intended recipient agent */
        reply_with:     /* sender's conversation identification key */
        in_reply_to:    /* recipient's conversation key */
        content:        /* message content */
        language:       /* name of language in which content is expressed */
        address:        /* sender's address */
        send_time:      /* time at which message is sent */
        receive_time:   /* time when message is received */
        M
)
```

Figure 4.2: The attribrites of the performative class

The first field of a performative message - its type - refers to the action the recipient is being asked to perform; this can be one of the following:

Message Type	Purpose
Accept-proposal	Used in the context of an existing dialogue to inform the recipient that a prior proposal has been accepted
Agree	Signals acceptance of facts asserted within the message
Cancel	Causes the recipient's *MailBox* and *ExecutionMonitor* to stop streaming reports to the originator
Cfp	Invites the recipient to make an offer concerning a specified resource
Confirm	Confirms the reservation of a resource
Disconfirm	Retracts a previously made conformation
Failure	Terminates a dialogue when the originator can no longer continue
Inform	Causes the recipient to add the associated content to its *Resource Database*
Inform-if	A conditional version of *Inform*

Inform-ref	Causes recipient to add some referenced content to its Resource Database
Not-understood	A more specific case of *Failure*
Propose	Encapsulates a proposal for achieving or buying a specified resource
Query-if	A conditional request for information
Query-ref	A request for information referenced by the message contents
Refuse	Terminates a dialogue, usually because the originator considers proposals have been unacceptable rather than unobtainable (cf. Failure)
Reject-proposal	Used in the context of an existing dialogue to inform the recipient that a prior proposal is unacceptable
Request	Asks the recipient to provide some resource or service
Request-when	A conditional version of *Request*
Request-whenever	A time conditional version of *Request*
Subscribe	Registers the originator, typically used to register with utility agents such as the Name-Servers and Visualisers

Table 4.1: The Message Types present within ZEUS Performatives

Performatives tend to originate from the states of dialogue nodes, (which are described in Section 4.2) or from the firing of rules that have a communication effect. Incoming performatives are then processed by the other component of the communication mechanism: the Message Handler, this is described next.

4.1.2 The Message Handler

The Message Handler is a ZEUS agent's internal mail sorting office, continually checking the incoming-message-queue of the Mailbox for new messages, and forwarding them to the relevant components of the agent. Its behaviour is controlled by two factors: first, whether a new message represents the start of a new dialogue or it is part of an existing dialogue; and second, on message processing rules registered in the Handler by other components of the agent.

New dialogues

For new dialogues, identified by messages with a null in-reply-to field in the message object, processing is governed solely by the rules registered with the Handler. The rules take two basic forms, object rules and engine rules:

Object Rules	Message-pattern → action-type object-reference method-name
Engine Rules	Message-pattern → action-type fully-qualified-graph-name

Where message-pattern is a partial description of a message performative that gets matched against the new incoming message object, object-reference is a Java object and method-name is the name of a public method of the object referenced by object-reference.

The action-type can be set to either EXECUTE_ONCE or EXECUTE_MANY. Whereas the rule is deleted from the Message Handler after execution in the case of EXECUTE_ONCE, it is retained after execution in the case of EXECUTE_MANY. The fully-qualified-graph-name is a string reference to the qualified name of one of the Co-ordination Engine graphs (described in section 4.2).

For object rules, new messages are matched against the message-pattern, with successful matches resulting in the method method-name of the object referenced by object-reference being invoked with the new message object as its input argument. The invocation mechanism is based on Java reflection. Object rules are intended to implement short-lived and simple reactive behaviour, for example automatic responses to requests for information.

For engine rules, a successful match of the incoming message against the message-pattern results in a call to the Co-ordination Engine to launch the graph referenced by the fully-qualified-graph-name with the message object as its input argument. Engine rules are intended for long-lived and/or complex behaviour such as requests to achieve a particular goal, which might lead to planning, negotiation with other agents and plan execution and monitoring. The generic ZEUS agent has some predefined object and engine rules for dealing with standard messages such as requests for information or requests to achieve goals.

Continuation dialogues

For continuation dialogues, the default behaviour of the Message Handler is to forward new messages to the Co-ordination Engine (which as described later, provides a mechanism for managing long-lived dialogues). However, this default behaviour can be overridden by object rules of the form

reply-message-pattern → action-type object-reference method-name

where reply-message-pattern is a message-pattern with a non-null value of its in-reply-to field. Thus, if a message matches the reply-message-pattern of a rule, then the rule is invoked as described earlier; otherwise, the default behaviour of forwarding the message to the Co-ordination Engine is applied.

Thus support for the declarative specification of behaviour in the Message Handler is provided through the use of pattern-action rules. This allows the processing behaviour of the Message Handler to be modified, even at runtime, simply by adding new processing rules or deleting existing ones.

4.2 THE CO-ORDINATION ENGINE

The role of an agent's Co-ordination Engine is to manage its problem solving behaviours, particularly those involving multi-agent collaboration. In addition to the general requirement for declarative specification of behaviour, the design of the Co-ordination Engine was governed also by the requirement that an agent should be capable of engaging in many tasks simultaneously. This meant that the Engine should support some form of multi-tasking. However, because of the costs involved, simple multi-threading was deemed inappropriate, since the number of independent tasks could potentially run into hundreds. Thus we choose to represent problem solving behaviours as recursive transition network graphs, which are interpreted by a recursive finite state machine.

Problem solving behaviour representation and processing

Figure 4.3 depicts a code fragment defining a simple graph and its equivalent pictorial representation. As the figure shows, behaviour graphs are specified as networks of nodes interconnected by directed arcs. The ZEUS representation of a graph is as a two-dimensional array of strings, where each string represents the fully qualified class name of the relevant node or arc.

The processing of a graph involves starting from a designated start node and attempting to traverse the graph until a terminal node is reached. The processing is controlled by a Graph class, which is the superclass of all behaviour graphs. The nodes of a graph implement the processing points, while its arcs implement tests to determine whether traversal from one node to another is valid. Each node and arc accept an input argument on which they act to return an output argument. Thus, information follows through a graph from node to arc, in tandem with the traversal path. To allow for recursion, graphs are themselves also arcs; for example, in Figure 4.3, the arc from S3 to S4 is a recursive invocation of simple_graph.

```
public class simple_graph extends Graph {
  static String[ ][ ] entry = {
      {"S1",
       "A1", "S2",
       "A2", "S3"},
      {"S2",
       "A3", "S4"},
      {"S3",
       "A4", "S3",
       "simple_graph", "S4"},
      {"S4"}
  };

  public simple_graph() {
    super(entry,"S1");
  }
```

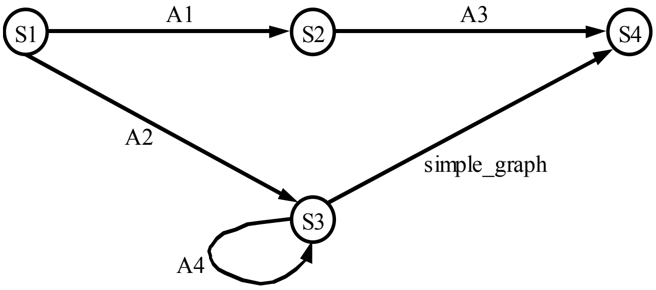


Figure 4.3: Behaviour graph representation — simple_graph.

All nodes of a graph must implement two functions: an exec() and a reset() function.

- The exec() function performs the core processing of the node, and returns one of the values — OK, FAIL or WAIT. A return value of OK indicates that the processing at that node has succeeded, whereas FAIL indicates failure. A return value of WAIT, which must be associated with a timeout value and/or a message-reply-key, indicates that processing of the node must be suspended until the timeout period expires and/or a message with the specified message-reply-key is received.
- The reset() function undoes any changes made on the input data by the exec() function — this is in order to support backtracking.

Arcs, on the other hand, must implement only a test() function that returns a Boolean value that indicates whether or not traversal of the arc is valid.

Multi-tasking

In order to allow for the parallel processing of multiple graphs, control of processing of graphs is managed by a priority FIFO node-queue. For example, processing of simple_graph of Figure 4.3 proceeds as follows: the graph is launched by creating its designated start node (S1) using Java's dynamic (runtime) object creation mechanism. Next, the input of the node is set to the argument passed with the launch command. Now, the node is queued onto the node-queue to await execution. Once the node is selected for execution, its exec() method is called, and if it returns OK, then the first arc emanating from the node (A1) is dynamically created. The arc is initialised with the output of S1, and its test() method called. If the test() method succeeds, S2 is dynamically created, initialised with the output of A1, and queued onto the node-queue. The use of the node-queue therefore allows many graphs to be executed simultaneously by interleaving node processing.

Parallelism

Further parallelism is supported by a mechanism whereby certain arcs or graphs can be designated as parallel graphs. For such an arc or graph, whenever its input is an array, then a (non-parallel) copy of the arc/graph is created for each element of the array. The copies are managed in a *k-out-of-n* fashion, i.e. the parent arc/graph is traversed if *k* or more of its child copies succeed. The value of *k* is specified in the definition of the parallel graph. The parallel graph mechanism is particularly useful during delegation or contracting, where many independent jobs may need to be contracted simultaneously.

Backtracking

To continue with our example, assume that when S2 is processed by calling its exec() method, it returns OK, however, when its first arc (A3) is executed it fails. In such a case, the processor attempts to backtrack by trying the next arc from S2. Since, S2 has no more arcs, its reset() method is called to undo any changes made by its exec() method. Next, its predecessor node (S1) is called to attempt traversing the graph by following its next arc (i.e. A2). Backtracking is also initiated whenever a node's exec() method fails.

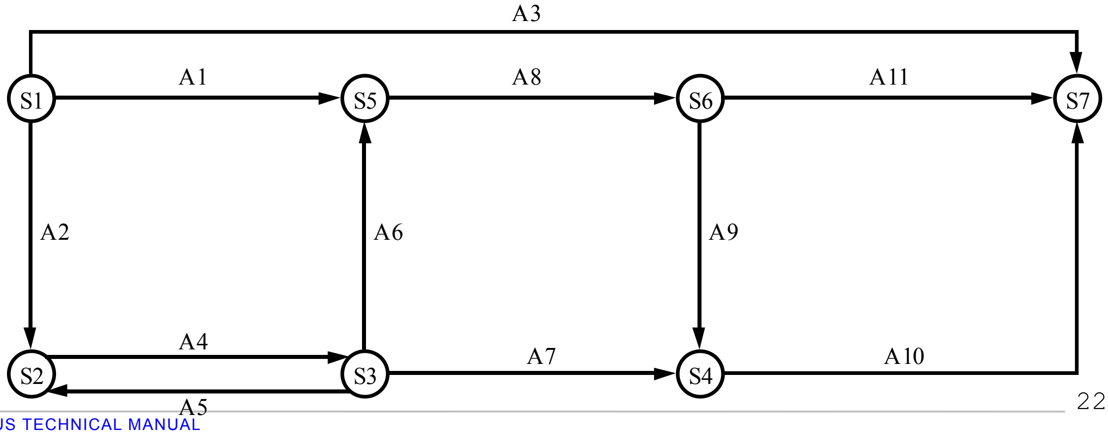
Communication

In the graph framework, support for inter-agent communicating is achieved through use of the WAIT return value of a node's exec() method. For example, a node engaged in communication would send out a message and then ask to be suspended by returning WAIT with an associated timeout value or message-reply-key. The node will then only be re-queued for processing when the timeout expires and/or a message with the required key is received.

The recursive transition network approach used to define behaviour satisfied our requirements for declarative specification of behaviour and support for multi-tasking. An alternative approach considered was rule-based processing; however this was rejected for a number of reasons. First, while rule-based systems allow declarative specifications and parallel processing, the management of contextual information (i.e. the data on which decisions and actions are based) becomes confusing when multiple independent behaviours act on the same data. Furthermore, this makes backtracking also difficult. A final consideration in its favour was that we also believe the transition network representation is more intuitive.

The Default Problem Solving Behaviour

The generic ZEUS agent comes equipped with a predefined goal-processing graph, this is depicted in Figure 4.4 and explained in Table 4.2. The graph controls basic problem solving behaviour for achieving goals, and can be viewed as logically composed of three phases — a resource allocation phase, a negotiation phase and a commitment management phase. In subsequent paragraphs, we shall describe the behaviour of the graph with an example.



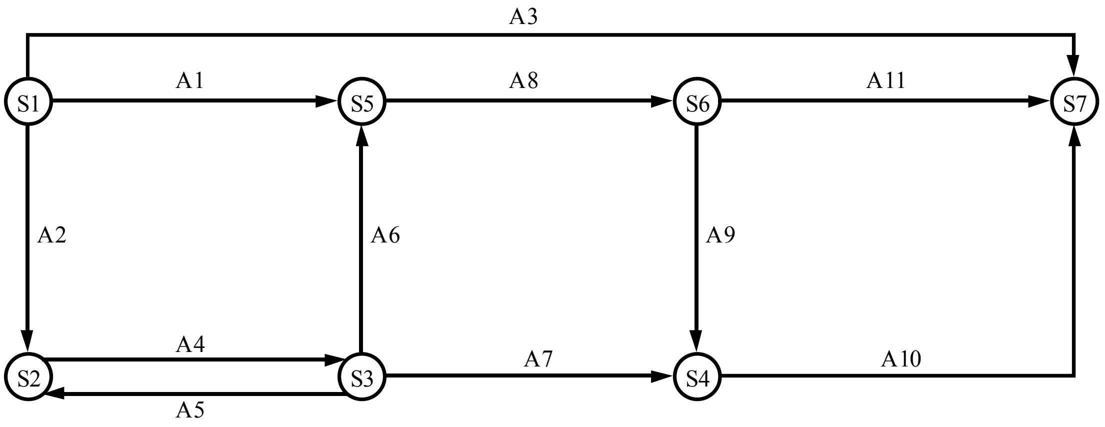


Figure 4.4: A simplified default goal processing graph (housekeeping nodes and arcs not shown).

Node/Arc	Description/Transition Condition	Phase
S1	Create and initialise data structure that holds goal contextual information Invoke Planner	Resource allocation
A1	No external involvement needed and agent is **not** initiator of goal	Resource allocation
A2	External involvement needed	Resource allocation
A3	No external involvement needed and agent **is** initiator of goal	Resource allocation
S2	Prepare to place external contracts	Negotiation
A4	Invoke initiator-side negotiation behaviour	Negotiation
S3	Resume planning with results of negotiation	Resource allocation
A5	Further external involvement needed	Resource allocation
A6	No further external involvement needed and agent is **not** initiator of goal	Resource allocation
A7	No further external involvement needed and agent **is** initiator of goal	Resource allocation
S4	Send confirmation and rejection messages to relevant contractor agents	Negotiation
A10	**True**	Commitment
S5	Prepare to negotiate with agent that request goal	Negotiation
A8	Invoke respondent-side negotiation behaviour	Negotiation
S6	Check that confirmation message has been received from the agent that requested goal	Negotiation
A9	Some external involvement was requested	Negotiation
A11	No external involvement was requested	Commitment
S7	Firmly commit to plan to achieve goal Set up monitors to manage plan execution	Commitment

Table 4.2: Descriptions of the nodes and arcs of Figure 4.4.

A Walkthrough of the Default Problem Solving Behaviour

Consider that a message is received by an agent to achieve a goal, x. Processing this message involves the agent's Message Handler sending a request to its Co-ordination Engine to launch the default goal processing graph with the new message as its input argument. Launching the graph involves the creation of an instance of its designated start node (S1 of Figure 4.4), and the node's input argument set to the new message.

When executed, S1 first creates a data structure to hold the contextual information that will be generated as a result of processing the goal. Next, the data structure is initialised with the goal's parameters (taken from the content of the message passed as input to the node). Now, the node calls the Planner/Scheduler to plan a sequence of actions to achieve the goal. If the Planner has no competence whatsoever in dealing with the goal then the node fails, which in turn will cause the graph to fail.

Assume, however, that the Planner plans a sequence of actions to achieve the goal, but requires that some subgoals y and z should be achieved externally by other agents – this may be because of lack of time, competence, information or other resources. Now, A2 is the only viable arc from S1, since its test condition that external collaboration is required (a non-empty list of external subgoals) is satisfied. Thus we arrive at S2, where the agent prepares to contract out the subgoals y and z; this is achieved by executing A4.

A4 is in fact defined as a 0-*out-of-n* parallel graph. Thus, if its input is an array, then a (non-parallel) child copy of itself is created for each element of the array. Therefore, with the sub-goals y and z as its inputs, independent child copies of A4 are created to handle each of y and z. Secondly, by being defined as 0-*out-of-n*, the parent A4 is traversed if zero or more of its child copies succeed. Thus, on contracting out y and z, the parent A4 will be traversed if none, one or both goals are contracted out.

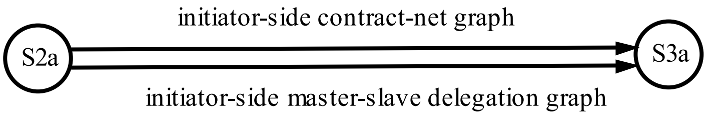

initiator-side contract-net graph
S2a
S3a
initiator-side master-slave delegation graph

Figure 4.5: The A4 parallel graph.

The graph defined by A4 (an example is shown in Figure 4.5) simply serves to provide a placeholder for the potentially many arcs/graphs that define the *initiator-side behaviour* of various negotiation protocols. It could include, for example, arcs/graphs defining the contract manager's behaviour at the request for proposals stage of the contract-net protocol, or the contract manager's initial delegation behaviour in the master-slave delegation protocol.

As the graph processor attempts to traverse the first viable arc from S2a to S3a, the choice of negotiation protocol depends on two factors. Firstly, the order in which the arcs from S2a to S3a are listed in the A4 graph specification, and secondly on whether the selected arc is indeed traversable in the given context. The backtracking capability of the graph processor ensures that if viable arcs exist from S2a to S3a, then at least one will be selected. In ZEUS, negotiation graphs (e.g. the initiator-side negotiation graphs in Figure 4.5) typically comprise two main parts: an *applicability assessment* section and a *negotiation dialogue* section. The applicability section simply checks whether the protocol is applicable in the given context, while the dialogue section actually implements the protocol.

Once A4 is traversed, at S3 the Planner is re-called with the results of the contracting process. Depending on its input, the Planner returns one of three possible results:

(a) that the planning process failed and no solution could be found for the original problem,

(b) that the planning process completed successfully and no further contracting is required, or

(c) it might return a new list of subgoals to be contracted out, as well as a partitioned list of prior subcontracts, some of which should be rejected and others that should be accepted.

If further contracting is required, A5 will return the goal processor to S2 to begin a new phase of contracting. If, alternatively there are no more subgoals to contract out and the planning process completed successfully, then A6 or A7 will be selected. Which one gets chosen depends on whether or not the agent is trying to achieve the original goal x for itself or on behalf of another agent. In the latter case, A6 is selected, which leads to S5.

At S5, some housekeeping functions are performed, in preparation for negotiation with the agent that re quested that goal x be achieved. The actual respondent-side negotiation is performed when A8 is executed. A8 is in fact a placeholder for potentially many arcs/graphs that define different *respondent-side negotiation* behaviour. Again the ordering of the arcs/graphs (that replace A8) and the context determine which negotiation protocol gets selected.

Following respondent-side negotiation (A8), at S6 checks are performed to determine that the agent has been awarded the contract, i.e. that an award confirmation message is received from the agent that requested goal x. Next, A9 or A11 is traversed, depending on whether or not the goal x had any external subgoals. Since in our example sub-goals y and z were contracted out, A9 is selected, leading to S4. At S4, award confirmation messages are sent out to the agents selected to perform the contracted subgoals, and rejection messages sent out to those agents that did not get selected. Finally, at S7, the plan constructed to achieve the goal x is scheduled for execution, and monitors are set up to manage the plan execution process.

From the foregoing description, a couple of points should be clear. First, that the transition network representation makes it relatively easy for one to redefine, if needed, the basic goal processing behaviour outlined above. However, we believe our conceptual decomposition of the goal processing process into resource allocation (planning), negotiation and commitment phases, and our default graph are fairly generic and applicable in a number of domains.

Secondly, the transition network approach makes it easy for negotiation protocols and strategies to be added to a ZEUS agent. Simply, initiator- and respondent-side negotiation graphs have to be defined for the protocol/strategy, and integrated into the default graph between S2a and S3a of Figure 4.5, and S5 and S6 of Figure 4.4, respectively. In fact, the generic ZEUS agent already has some predefined negotiation protocols such as contract-net, master-slave delegation and some auction protocols. Figure 4.6 and Table 4.3 describe our current initiator- and respondent-side implementations of the contract-net protocol. In typical negotiation graphs, the negotiation strategy logic is defined within the exec() methods of the nodes in the graphs. In some cases, this may even involve a call to external programs. The organisational relationships defined by Agent Component Library are typically used in the applicability test portions of negotiation protocol graph specifications. For example, the applicability test portion of the initiator-side contract-net graph of Figure 4.6 mandates a preference for co-worker agents before peer agents when contracting out tasks (by the ordering of the arcs A1 and A2, where A1 precedes A2).

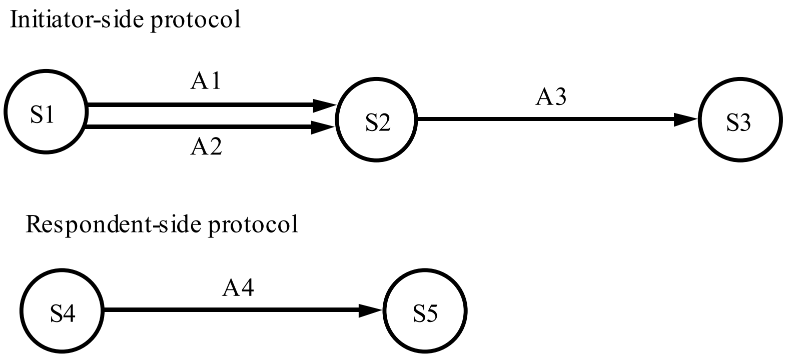


Figure 4.6: Sample ZEUS implementation of the contract-net protocol

Initiator-side contract-net behaviour	
Node/Arc	**Description/Transition Condition**
S1	Identify agents that can perform goal
A1	Select subset of agents that can perform goal and who are co-workers (check ≠ ∅)
A2	Select subset of agents that can perform goal and who are peers (check ≠ ∅)
S2	Send request for proposals to selected agent and await responses
A3	Check that an **accept** response has been received

S3	**Done**

Respondent-side contract-net behaviour	
Node/Arc	**Description/Transition Condition**
S4	Evaluate cost Send **accept** message Await response
A4	Contract award message received
S5	**Done**

Table 4.3: Descriptions of the nodes and arcs of Figure 4.6.

The Default Rationality Model

It is worth briefly commenting on the rationality model implicit in the Co-ordination Engine and the default goal-processing graph. Used as is, the default goal processing graph implicitly implements the following:

(a) an agent would accept any goal for which it has the available resources to pursue,

(b) the agent will continue accepting goals on a first-come first-served basis until its capacity to do so is exhausted, and

(c) once an agent has accepted a goal, the agent is fully committed to the goal, and will do all in its power to ensure the goal is achieved. (As we shall see when discussing exception handling, the agent remains committed to an accepted goal even when it might no longer be in its best interest to remain so[2]).

It is worth noting that the current version of the Co-ordination Engine has no explicit rationality model, i.e. there is no reasoning framework to determine from whom, and when to accept goals, or equivalently, when to abandon goals although they may be technically achievable. Such a model, if needed, can be implemented by adding an agenda (and associated reasoning rules) to control (i) the conditions under which the goal processing graph is launched — for goal selection, and (ii) the node-queue of the Co-ordination Engine — for goal scheduling and abandonment control. The reasoning rules of the agenda should be essentially equivalent to the desire filter of the belief-desire-intention agent architecture [7]. Given the application domains envisaged for ZEUS, such an explicit rationality model was not deemed particularly necessary, although one is likely to be included in future versions of the system.

4.3 The Planner and Scheduler

The role of the Planner/Scheduler is to construct action sequences that achieve desired input goals. The Planner is under the control of the Co-ordination Engine, which initiates planning and also manages the contracting of any subgoals that the agent cannot achieve.

Plan Operator Representation

Planning operators (actions or tasks) are represented in the classical fashion as primitive or summary operators. Primitive operators are defined in terms of their preconditions, effects, cost, duration and constraints and precondition order, while summary operators are defined in terms of an expansion or mini-plan (of existing primitive tasks).

[2] This reflects practice in commercial domains, where for legal reasons and also in order to maintain customer confidence, contracts that are later found to be non-profitable are not automatically abandoned.

For planning efficiency, the preconditions and effects of operators can be qualified, just as variables and methods can take modifiers in programming languages). Preconditions can be marked as:

- read-only, implying the condition is not consumed by the operator on execution, but simply read.
- local, implying that the Planner can only apply the operator if it possesses the resource, or it can produce the resource on its own. This means the agent cannot delegate production of the required resource to another agent.

Effect conditions can be marked as either public (the default) or private. The Planner will select an operator to produce a desired effect only if that effect is marked public. Hence side-effects should be marked as private to prevent the Planner from selecting the operator in order to produce such effects. A good example is "lying on the ground" is a side-effect of hitting someone, but if the intention is to merely to instruct someone to lie on the ground, hitting them is a far from ideal way of achieving it.

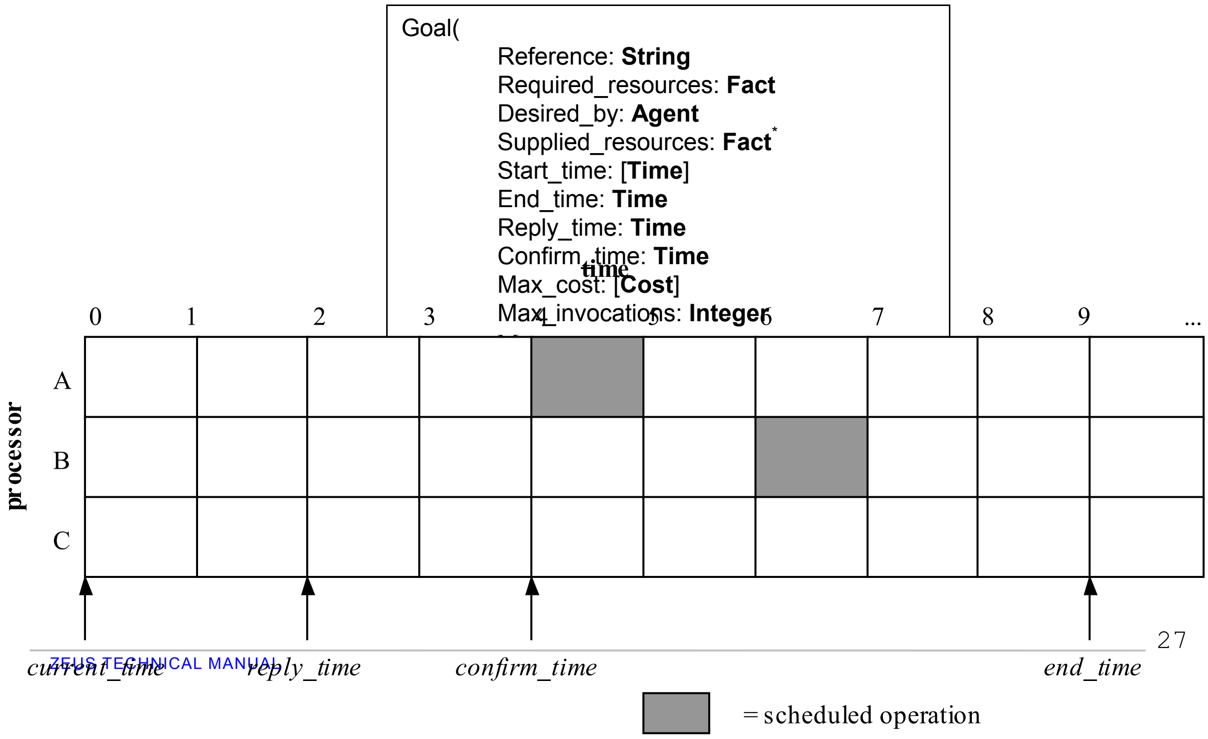
Multi-agent Planning with Primitive Operators

The Planner utilises classical partial order means end planning in its plan construction process. (For a review of the planning literature see [8]). Thus, given a goal of the form of Figure 4.7(a), the Planner searches its Plan Database for an operator with a public effect that unifies (with unification bindings θ) with the desired_effect of the goal. If multiple operators are found, they are ranked by cost, and then by duration. Next, the Planner selects the first ranked operator, constrains its preconditions and effects with θ, and then attempts to schedule the operator into its diary. If the operator cannot be scheduled, the Planner backtracks and repeats the process with the next applicable operator.

The Planner's diary is a two-dimensional array, with time on one dimension and processors on another, as depicted in Figure 4.7(b). This approach (also used in job-shop scheduling), uses the principle that an agent will schedule its activities over a finite time period defined by the length of its diary, and across a finite number of processors, given by the width of the diary. In the ZEUS implementation, processors are independent task execution threads.

The scheduling of tasks into the plan diary is constrained by the end_time, the reply_time and the confirm_time of the goal. The end_time specifies the latest time by which the desired_effect should be achieved; while the reply_time specifies the latest time by which the agent planning the task must inform the agent requesting the task whether it can perform the goal. The confirm_time specifies the latest time by which the requesting agent will inform the performing agent whether it has been awarded the contract.

Thus, operator scheduling starts from the end_time and progresses towards the confirm_time, utilising any free processors. Furthermore, the entire planning process must be completed (or else aborted) by the reply_time. (The start_time is used in instances where a service agreement is being created. In such cases, one agent is requesting that another agent plan a task that will be executed *n* number of times at some future date, between some start_time and some end_time.)



Goal(
Reference: **String**
Required_resources: **Fact**
Desired_by: **Agent**
Supplied_resources: **Fact***
Start_time: [**Time**]
End_time: **Time**
Reply_time: **Time**
Confirm_time: **Time**
Max_cost: [**Cost**]
Max_invocations: **Integer**

Figure 4.7: (a) The goal data structure. (b) A plan diary.

If an operator is successfully scheduled into the plan diary, then the Planner will attempt to anchor all the preconditions of the operator, in the order mandated by the operator's precondition ordering constraint. A precondition can be anchored in one of four ways:

(a) a resource in the Resource Database that unifies with the precondition can be reserved for it, or

(b) a matching unused effect (i.e. a side effect) of a previously scheduled operator can be reserved for it, or

(c) an operator that produces the required precondition can be scheduled into the diary, or

(d) an external agent can be found to produce the precondition. In the latter three cases, care must be taken to ensure that the required resource is produced before the operator needs it.

The anchoring process can be illustrated with an example; assume an operator P has preconditions a, b, c, and d, and their order constraint enforces the ordering [a] → [b, c] → [d]. Thus, firstly, the resource a is checked against the Resource Database. If a resource a* is found in the database that unifies with a (with bindings φ), then a* is reserved for a. The reservation process takes into account whether or not the operator P will consume the resource, and if it does, when that is scheduled to happen. Thus, multiple read-only preconditions can reserve the same resource.

Once a reservation is made for the resource a, the other preconditions and the effects of the operator P are constrained with the bindings φ, and the next set of preconditions in the sequence, b and c, checked against the Resource Database. Assume however that no resources matching b or c could be found in the database. Next, the Planner checks its diary for any scheduled operators that produce b* or c* as unused side effects, and which are scheduled to complete execution before the operator P is scheduled to start execution. Assume that a scheduled operator is found that produces b* within the required time window, but none for c*. Now, b* is unified with b, and the unification bindings used to constrain further the operator P. Also, the effect b* is reserved for b. To deal with the precondition c, the Planner searches its Plan Database for any operators that produce c* as one of their public effects. If some operators can be found, the planning process proceeds recursively as described above. If however, no applicable operators are available and the precondition c is not marked as local, then the planner creates a new subgoal to achieve c and adds the subgoal to a list of subgoals that must be achieved by other external agents. If in the sequence of preconditions [a] → [b, c] → [d], c must precede d, and c has to be achieved externally, then planning is suspended for d until c is successfully contracted out.

Thus, the planning process proceeds in the **backward chaining** manner described until one or more of the following conditions apply:

(i) the scheduler runs out of available processor space;

(ii) all the preconditions in the plan either (a) have been allocated resources from the Resource Database, or (b) have been allocated side effects of previously scheduled operators, or (c) external subgoals have been created for them, or (d) plan operators have been scheduled to produce their required conditions;

(iii) the planning process has been suspended until some external subgoals have been contracted out.

In any case, if a partial plan has been constructed, the planner will return to the Co-ordination Engine a list of subgoals that must be contracted out to other agents. Once the results of the contracting process are returned to the Planner, it utilises them to progress the planning process by backtracking on plan branches with subgoals that could not be contracted out, and elaborating suspended branches that depended on successfully placed external contracts. The planning process only successfully terminates when there are no unanchored preconditions in the plan.

Multi-agent Planning with Summary Operators

The plan construction process with summary operators differs somewhat from the case with primitive operators. For example, consider that the summary operator of Figure 4.8 is selected for inclusion in a plan in order to produce the effect g. The Planner traverses the operator specification from right to left, following its precondition-effect links, and attempts to replace each node with a concrete (i.e. eventually primitive) operator. Thus, it starts with node_4, and searches its Plan Database for other summary or primitive operators that have preconditions and effects that are supersets of the node's preconditions and effects, respectively.

Next, it ranks the applicable operators found and then selects the highest ranked one for inclusion in the plan; (the ranking criteria being by summary operators first, then by cost, and finally by duration). If the selected operator is a summary operator, then this planning process is repeated recursively on it. If, however, the selected operator is primitive, then the Planner attempts to schedule the operator into its plan diary.

Once all the nodes in a summary operator have been replaced by primitive operators (either directly or by recursively expanding other child summary operators), the Planner then attempts to enforce the effect-precondition links between the nodes on their concrete images in its diary. This serves to anchor some of the preconditions of the concrete primitive operators that form the expansion of the summary operator. For example, the effect-precondition link between node_2 and node_4 of Figure 4.8 anchors the precondition, e of node_4. Finally, the Planner attempts to anchor all remaining unanchored preconditions in the expansion; e.g. the precondition, b of node_2 or a of node_1.

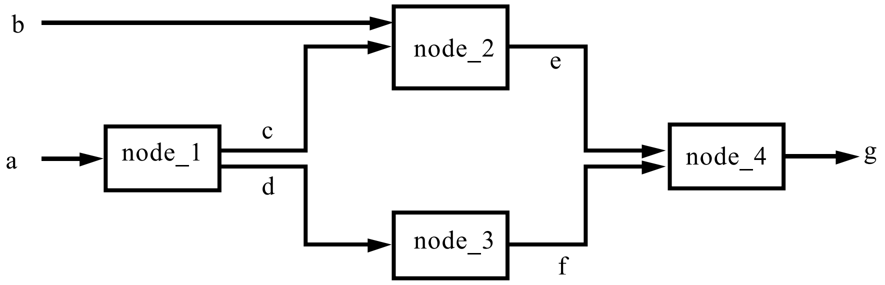


Figure 4.8: An example summary operator.

The summary operator expansion mechanism described above is sufficient when a single agent is solely responsible for all operators in a plan. When multiple agents are involved, care has to be taken to ensure the correct routing of results that are produced by one agent but required by another. For example, if during expansion of a given node of a summary operator there are no operators in the Plan Database that have preconditions and effects that match those of the node, then the Planner has to create a subgoal to achieve the required effects of the node externally. For instance, if on expansion of node_4 of the summary operator of Figure 16 no operators can be found in the Plan Database that produce the effect, g, given inputs, e and f, then an external subgoal to produce, g, given e and f is created. Note that in the case of subgoals for primitive operators, the subgoal statement was simply of the form *produce* α, whereas for summary operator nodes, subgoal statements are now of the form *produce* α *given* β. Assume that the subgoal to produce g, given e and f, is successfully delegated to some agent A. Further, assume that on expansion of node_3, again no operator could be found that produces the required effect, f, given input, d. Again, assume the subgoal for this problem is successfully delegated to some other agent B. Now, the planner has to ensure that agent B has the required precondition, d, for its portion of the task, and further, that agent B sends its result, f, to agent A, so that agent A can perform its portion of the task.

Managing Resource Reservation Conflicts

The planner possess a mechanism to ensure the coherence of produced and supplied items when subgoals of the form *produce* α *given* β occur within multi-agent problem solving. Typical conflict situations take one of two forms. In the first case, consider that an agent A delegates to another agent B the task to produce x given y and z. Now, in creating a plan to produce x, agent B makes a reservation of the expected supplied item y. At this point in its planning process, agent B has not reserved the expected input item z, and it cannot say whether it may need it in the future. Further, as part of the plan to produce x, agent B needs to delegate to agent C a task to produce u, given v. But, the item z is still available and conceivably, agent C might need it whereas agent B might not. Thus, agent B delegates to agent C the task to produce u, given v and z. Now assume that in its plan to produce u, agent C reserves both v and z. This makes z unavailable to agent B, and agent B must be notified of this. Further, agent A must be notified to supply the input y to agent B and z to agent C (although agent A has no direct contract with agent C).

In conflicts of the second form, consider that an agent A delegates to another agent B the task to produce x given y and z. In creating a plan to produce x, agent B decides to delegate to agent C the task to produce u and to agent D the task to produce v. At this point in its planning process, agent B does not need the expected inputs y or z, and it cannot say whether it may need either of them in the future. So, because agents C or D might conceivably need y or z, agent B decides to delegate to agent C the task to produce u, given y and z, and to agent D the task to produce v, given the same y and z. A conflict emerges if both agents C and D reserve the same resource y, for example. It is the responsibility of agent B's Planner to check the reservations made by both agents C and D to ensure there is no conflict. If a conflict is detected, only one of agents C or D is selected to perform its portion of the task, and the other portion delegated again with a revised list of supplied items.

4.4 EXECUTION MONITORING

Once a plan is constructed and scheduled for execution, each operator in the plan is executed in order at the operator's scheduled execution time, or alternatively, before the scheduled execution time if there is a free processor available. Operator execution, which is controlled by the Execution Monitor component, involves an invocation of the domain function specified in the operator's specification. The domain function, which is typically some legacy process, is invoked with the operator's preconditions as its input arguments, and it is expected to return the declared effects of the operator. The relevant output of the domain function (i.e. the operator's effects) are then passed as input to the appropriate downstream operators in the plan sequence.

Plan Execution and Exception Handling

An important role of the Execution Monitor is to detect failure during the scheduled execution of a plan. This can occur for a number of reasons:

(i) a resource reserved by an operator might be deleted,

(ii) an operator might begin execution but fail to complete because of insufficient scheduled time or some other reason,

(iii) an operator might successfully complete execution but return the wrong or incomplete results, or

(iv) some promised resource from another agent might not arrive on time (or alternatively, the other agent might notify our agent that it can no longer provide the promised resource). In either case, the net effect is that some precondition of an operator in the plan would lose its anchor.

In the case of the latter, if the Planner deems that it has sufficient time to replan before the whole plan goes out of schedule, then it initiates replanning. Replanning essentially takes advantage of the Planner's normal planning and backtracking mechanisms. The Planner attempts to anchor the dangling precondition in one of four ways, either by:

(i) allocating a new resource to it – if one can be found, or

(ii) creating a new plan to produce the required resource, or

(iii) creating a sub-goal to produce the required resource and contracting out the subgoal, or

(iv) backtracking on the plan branch containing the dangling precondition.

In the latter three methods, the replanning process may involve the placement of new contracts with other agents. Furthermore, in the fourth method, some existing contracts might need to be cancelled if the new plan arrived at by backtracking no longer requires the results of those contracts. If all four approaches fail, in which case the Planner cannot devise any plan to achieve the desired goal, it will try to contract out the original root goal itself. If this also fails, the Planner is forced to report failure to the agent that originally requested the root goal. It is worth noting here that the whole process of replanning and the potential costs of placing new contracts and cancelling existing ones might end up being very unprofitable for the agent.[3]

Tables 4.4 and 4.5 present an example scenario illustrating some of the mechanisms involved in the excep

[3] In future versions of the system, we plan to introduce contract cancellation penalties that the agents can utilise to determine whether or not to replan.

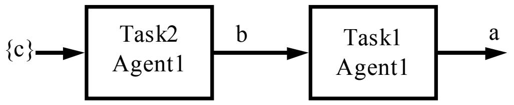
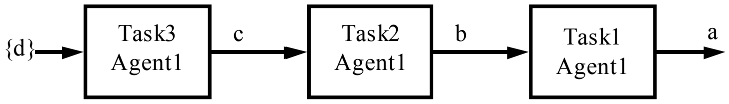
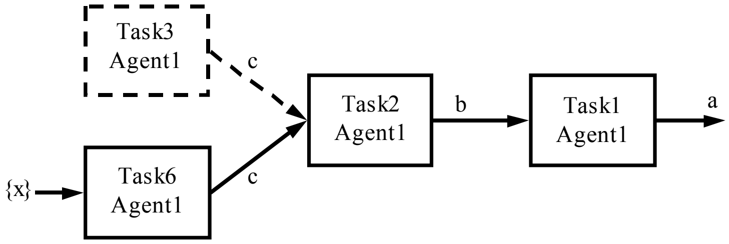
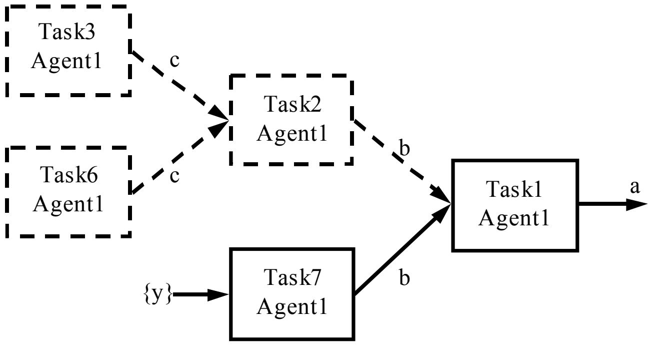
tion handling process (although not included are situations where existing contracts need to be cancelled).

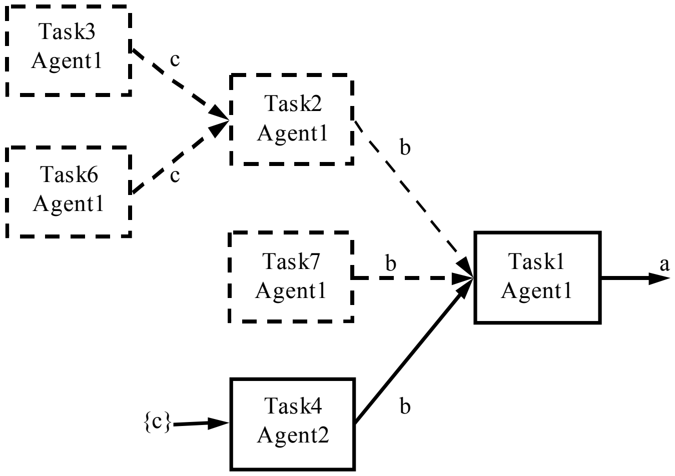
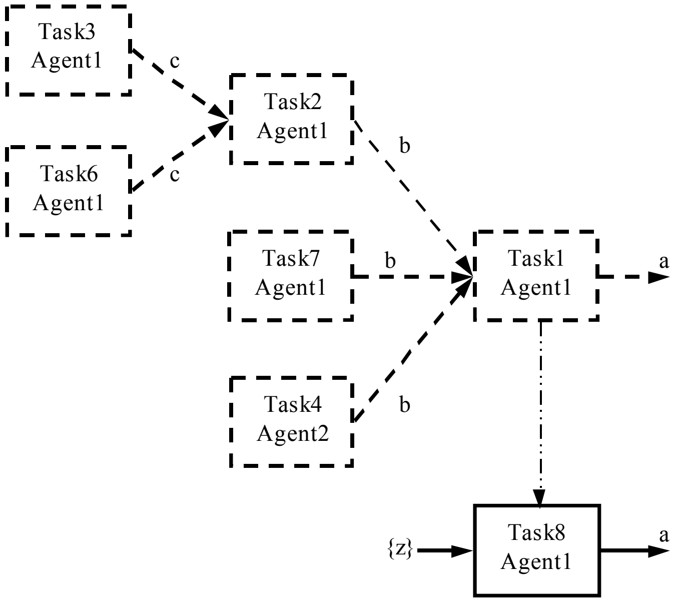
	Agent1	Agent2	Agent3

Task Database	{b}→Task1/10→{a}
	{c}→Task2/20→{b}
	{d}→Task3/30→{c}
	{x}→Task6/60→{c}
	{y}→Task7/70→{b}
	{z}→Task8/80→{a}
		{c}→Task4/40→{b}	{b}→Task4/40→{a}

Resource Database	{c,d,x,y,z}	{c}	{b}
Key	{Preconditions}→TaskID/Cost→{effects}		

Table 4.4: Initial states of the agents involved in the exception handling scenario described in Table 4.5.

Action/Comments	Results
Achieve Agent1 a The agents' planners use a best-first selection policy with no lookahead – thus given the goal to achieve a, the cheapest applicable task (Task1) is selected first.	
Delete Agent1 c Deletion of the database resource, c, leads to a new operator (Task3) to anchor the precondition of Task2.	 {d} Task3 Agent1 c Task2 Agent1 b Task1 Agent1 a
Delete Agent1 d Deletion of the resource, d, means Task3 loses its anchor. Hence, the planner backtracks, and tries an alternative mechanism for achoring the precondition, c, of Task2.	 Task3 Agent1 c Task2 Agent1 b Task1 Agent1 a {x} Task6 Agent1 c
Delete Agent1 x Again, more backtracking.	 Task3 Agent1 c Task2 Agent1 b Task6 Agent1 c Task1 Agent1 a {y} Task7 Agent1 b

Delete Agent1 y Deletion of the resource, y, leads to further backtracking and finally an external contract placed with Agent2. The best-first selection policy implies that once Task1 had been selected to achieve a, all possible means to anchor its preconditions will be tired, even if this involves an external contract. So, an external contract is placed to anchor the precondition, b, although the agent could have achieved the goal on its own if it used Task8 to achieve a.	 Task3 Agent1 c Task2 Agent1 Task6 Agent1 c b Task7 Agent1 b Task1 Agent1 a {c} Task4 Agent2 b
Delete Agent2 c Deletion of the resource, c, in Agent2, leads to the exception cascading back to Agent1, which now tries a completely different means to achieve the goal, a.	 Task3 Agent1 c Task2 Agent1 Task6 Agent1 c b Task7 Agent1 b Task1 Agent1 a Task4 Agent2 b {z} Task8 Agent1 a

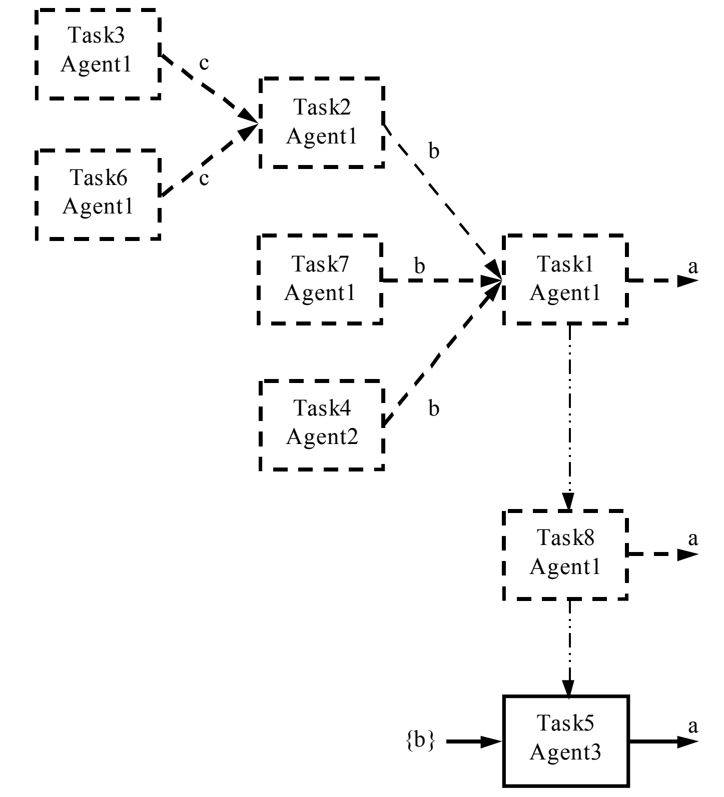
Delete Agent1 z Agent1 has now completely exhausted all possible ways of achieving the goal, a. However, its commitment to the contract forces it to contract out the entire goal to Agent3.	

Table 4.9: An exception handling scenario.

4.5 INTEGRATING ZEUS AGENTS WITH EXTERNAL PROGRAMS

The preceding sections have mentioned the primary interfaces between ZEUS agents and external programs. These include the domain functions in primitive plan operator specifications, and user-defined Co-ordination Engine graphs whose nodes make direct calls to external programs. Also mentioned was the fact that the Resource Database implements an interface that could be used, for example, with the database connectivity API of Java to link to proprietary databases – thus extending its potential size and functionality. For routine problem solving within the declared scope of the ZEUS toolkit, it is expected that for the most part, these primary mechanisms will suffice. However, the Agent Component Library also provides a secondary, more sophisticated interface, although employing it requires significant user programming. This done via a ZeusExternal interface class and an agent internal event model.

The ZeusExternal interface class allows users to link an external Java class (that implements the interface) to an executing ZEUS agent program. Once linked to the agent program, the external code can utilise the agent's public methods to query or modify the agent's internal state. Thus, for example, the resource and/or plan databases can be queried or modified.

The internal event model provides a mechanism whereby all significant events occurring in the agent can be monitored, e.g. planning events, resource, acquaintance and plan database events, message events, execution monitor events, and co-ordination engine events. The internal event model is similar to the Java event model, whereby objects register an interest in receiving events of a particular type, and all subsequent events of that type are forwarded to the object. So, using the event model, an external program that is linked to a ZEUS agent can monitor particular events in the agent and react to them, even by going so far as to use the agent's public methods to make changes to its internal state. Although the reader is forewarned that ZEUS currently does not even attempt to manage any internal inconsistencies that might result from such actions.

The **ZEUS Application Realisation** guide provides several examples of how to use the event model.

5 CONCLUSIONS

This document is still incomplete, but hopefully it should have outlined the philosophy and functionality underlying the ZEUS toolkit. It is hoped that the separation of the agent-level issues from the domain-level issues, will enable developers to create working multi-agent systems more easily and efficiently. If so, our decision to employ a declarative specification of behaviour that is interpreted by the generic ZEUS agent will have been vindicated.

Although this document provides an insight into how agent applications can be built from the toolkit components, it has deliberately avoided discussing how agent applications should be designed. There is also no description of how tools such as the ZEUS Agent Generator tool can be used to create executable applications. Instead, these are subjects dealt with in the ZEUS Methodology documents.

REFERENCES

1. Newell, A. (1982), The Knowledge Level, *Artificial Intelligence* **18**, 87-127.
2. Finin, T. & Labrou, Y. "KQML as an Agent Communication Language". In *Software Agents*, J.M. Bradshaw, (ed), MIT Press, Cambridge, Mass. 291–316, 1997.
3. The Foundation for Intelligent Physical Agents. "The FIPA'97 Specification". http://drogo.cselt.it/fipa/spec/fipa97/fipa97.htm
4. Genesereth, M.R. & Fikes, R.E. (1992) (Eds), "Knowledge Interchange Format", version 3.0 Reference Manual, Computer Science Department, Stanford University, Technical Report Logic-92-1, also available from http://www-ksl.stanford.edu/knowledge-sharing/kif.
5. Davis, R., Smith, R.G. Negotiation as a metaphor for distributed problem solving. Artificial Intelligence 20 (1983) 63–109.
6. Searle, J.R.: Speech acts. Cambridge University Press, 1969, Cambridge MA.
7. Rao, A.S. & Georgeff, M.P. "BDI agents: from theory to practice". Proc. 1st Int. Conf. on Multi-Agent Systems (ICMAS-95), San Francisco, June 1995, pp. 312–319.
8. Hendler, J., Tate, A. & Drummond, M. "AI Planning: systems and techniques", AI Magazine, May 1990.