Abstract

The multi-agent systems approach of knowledge level co-operation between autonomous agents promises significant benefits to distributed systems engineering, such as enhanced interoperability, scalability and re-configurability. However, thus far, because of the innate difficulty of constructing multi-agent systems, this promise has been largely unrealised. Hence, there is an emerging desire amongst agent developers to move away from developing point solutions to point problems in favour of developing methodologies and toolkits for building distributed multi-agent systems. This philosophy led to the development of the ZEUS Agent Building Toolkit, which facilitates the rapid development of collaborative agent applications through the provision of a library of agent-level components and an environment to support the agent building process. The ZEUS toolkit is a synthesis of established agent technologies with some novel solutions to provide an integrated collaborative agent building environment.

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, there is an urgent need for frameworks, methodologies and toolkits that support the rapid development of multi-agent systems [28]. Hence, the recent explosion in the number of multi-agent system development environments, frameworks and toolkits [2,4,5,7,8,22,25,34,35]. This paper describes 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 [1] source code of the agents.

The breakdown of the rest of this paper is as follows: Section 2 briefly describes the issues involved in building distributed multi-agent systems, emphasising the main problems raised by knowledge-level collaboration between heterogeneous, distributed and independently autonomous software systems. In Section 3, 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 4, we describe the ZEUS toolkit architecture; this section provides an overview of the design of the entire toolkit. We describe its Agent Component Library, the generic ZEUS agent, the ZEUS agent building approach, and a visual agent creation environment that supports the approach. Also, we briefly describe the suite of ZEUS utility agents including nameserver, facilitator and visualiser agents. In Section 5, we illustrate the multi-agent system creation process using the ZEUS toolkit with an example problem in supply chain provisioning. Section 6 describes some further design and 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. The reader not interested in implementation details may skip this section of the paper. Sec-
Section 7 discusses the main strengths and limitations of the ZEUS system, and situates our work by reviewing it within the context of other related work. Section 8 concludes the paper.

2 ISSUES INVOLVED IN BUILDING DISTRIBUTED MULTI-AGENT SYSTEMS

In this section, we present the various issues involved in building a toolkit for constructing a system of distributed and collaborating agents, and suggest that it is a non-trivial exercise. We begin by summarising why agents typically need to collaborate with other agents in order to solve problems, as is the case in multi-agent systems. Further, we consider the implications on the agents if such collaboration were to proceed at the knowledge level [30]. Next, we briefly review typical solutions to the problems posed by knowledge level collaboration. Finally, we conclude the section by examining how the solutions to the knowledge level collaboration problem impact on software engineering of collaborative multi-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, 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 is a context diagram illustrating the interplay between the various issues and their associated solutions.
Figure 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.

Information discovery 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, nameservers and facilitators might be arranged in hierarchies similar to that of Internet domain nameservers. Sometimes facilitators are also used to manage the message traffic within and between agent societies [25].

Communication: a few languages have been developed to meet the need for an agent-independent agent communication language (ACL). Notable examples include the Knowledge Query and Manipulation Language (KQML) [11] and FIPA ACL [39]. Most agent communication languages are based on speech act theory [36], 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 or 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. Furthermore, 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) [14], typically used with KQML, and FIPA SL [39] the preferred content language for use with the FIPA ACL.

Ontology: 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. Whilst some general-purpose ontologies have been proposed for use in inter-agent communication, like Cyc [23], a trend is towards the provision of editors for creating domain-
specific ontologies and inter-ontology translators (e.g. [16]), as is the case with the interlingua approach represented by KIF [14]. The argument being that, firstly, most general-purpose ontologies, e.g. Cyc, are unlikely to include the intricacies of all possible domains; and secondly that they are likely to be very large and unnecessarily complex for most applications. Given a shared ontology for a multi-agent system application, there still remains the problem of translating between the internal knowledge representations of each agent and the common ontology. Huhns et al. [18] use so-called articulation axioms, logical constraint expressions, to provide this mapping.

**Co-ordination:** 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. This is typified by client-server systems, but also includes systems where agents engage in activities according to their roles, thus co-ordination is implicitly performed by assigning a role to an agent. Wiederhold [41] advanced the use of mediators\(^1\) for co-ordinating multi-agent systems. Mediators are similar to facilitators but include the functionality of intra- and inter-society co-ordination. Their design and behaviour is domain and application dependent, and relies on them knowing *a priori* the abilities of the agents they control as well as the inter-relationships between these agents. In some applications, mediators also perform a conflict resolution role, and in others they contain the articulation axioms for translating agents’ internal knowledge representations into the shared ontology.

Contracting as a co-ordination mechanism is typified by the classic contract-net protocol of Davis & Smith [10], 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. The simplicity of the schemes makes it one of the most widely used co-ordination mechanism with many variants in the literature. Some interesting alternatives/variants to the contract-net protocol include various auction protocols such as the *english*, *dutch* and *double* auctions [32].

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. Nwana et al. [32] presents a gentle introduction to the literature on co-ordination of multi-agent systems.

**Integration with Legacy Software:** bluntly, agents do not solve the legacy software problem. Hence, they must be able to interact with legacy software. Genesereth & Ketchpel [13] discuss the problem of integrating legacy software with agent systems, and suggest three possible solutions to the problem. Firstly, rewriting the software — a costly approach. Secondly, through use of a separate piece of software called a *transducer* that acts as an interpreter between the agent communication language and the native protocol of the legacy system. Lastly, is the *wrapper* technique where the legacy program is augmented with code that enables it to communicate using the inter-agent language. Nwana & Wooldridge [31] provide a gentle introduction to software agent technologies including ACLs, ontologies, legacy systems integration issues, and other computing technologies that support agent systems development.

From the foregoing discussion it is clear that building effective multi-agent systems requires the integration of techniques from many AI and computing disciplines. Further, in addition to the scientific challenges associated with integrating such disparate techniques, there are also technological challenges related to building interacting distributed systems, hence, the difficulty in progressing the multi-agent paradigm into mainstream software engineering.

However, the problems and solutions outlined above are primarily concerned with managing collaboration, and have little to do with the domain and application specific aspects of the problem solving. Thus, most of the tech-

---

\(^1\) Some authors use facilitators and mediators interchangeably, while for others facilitators are ‘yellow pages’ facilities only, with no co-ordination role to which mediators typically allude. In this paper, we adopt the definition of facilitators as ‘yellow pages’ facilities only.
niques should be re-usable across domains with little modification. In this vein, it should be possible to ‘compo-
ponentise’ the agent level issues concerned, inter alia, with managing collaboration, such that they are available for
re-use by agent system developers. This way, developers can concentrate on implementing the domain and ap-
lication specific behaviour required of their agents. This idea formed the broad goal and hypothesis of the
ZEUS project. In the next section we elaborate on the aims, philosophy and assumptions that underpinned the
project.

3 AIMS, PHILOSOPHY AND ASSUMPTIONS OF THE ZEUS PROJECT

The aim of the ZEUS project was to facilitate the rapid development of new multi-agent applications by ab-
stracting into a toolkit the common principles and components underlying two collaborative multi-agent systems
we had developed for business process engineering [33] and multimedia information management [40]. 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 sys-
tems. Our design philosophy required that the toolkit encapsulate the following principles:

• Firstly, it 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 exe-
cution and monitoring, exception handling, etc. while the former covers the acquisition, representation and
use of domain-specific knowledge in problem solving. Thus, it was required that the toolkit provide agent
developers with all the agent-level functionality such that they need only provide the code implementing the
domain-specific problem solving abilities of the agents they define.

• Secondly, use of the toolkit should be based on the direct manipulation metaphor [37] whenever feasible.
This human-computer interaction metaphor is epitomised by the visual programming paradigm and the
‘pick-and-choose’ principle. That is, it was required that the toolkit support the agent creation process by
providing structured menus so that application developers could configure the functionality and modalities
required of their agents by selecting appropriate menu items.

• 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 also required that the toolkit utilise ‘standardised’ technology wherever feasible, or be de-
signed with standardisation in mind. We argue in Ndumu & Nwana [28] that standardisation is sine qua non
for the industrial uptake of this technology. This way, we envisaged components which could later be re-
placed with little difficulty by ‘more standardised’ components. Standardisation in agent applications ad-
dress agent-agent, agent-legacy software and agent-user interactions using a standardised ACL for example,
and there have been attempts to standardise the content languages of ACLs too.

3.1 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:

• that it should provide predefined information discovery agents such as nameserver and facilitator
agents, and
• that it should also provide extensive facilities for visualising and debugging societies of ZEUS agents.
3.2 ZEUS Agent Assumptions and Typical Application Domains

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.

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.

4 THE ZEUS TOOLKIT ARCHITECTURE AND APPROACH

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 2: 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.
4.1 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 2 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 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 2 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.
4.1.1 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 shows the architecture of the generic ZEUS agent that is not too dissimilar from other collaborative agent architectures in the literature.

As Figure 3 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.

Figure 3: Architecture of the generic ZEUS agent
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.

### 4.1.2 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.
4.1.3 Summary

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. Finally, co-operative problem solving between agents in task-oriented domains clearly requires planning and scheduling capabilities. Hence, the reader can see how our agent design derives from the principles spelt out in Section 2. In Section 6 we describe the design and implementation of some main components of the Agent Component Library.

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

4.2 The ZEUS Agent Building Approach and Environment

In the previous section, we looked at the main components of the Agent Component Library, and in the use case scenario, described how the components work together in the generic ZEUS agent architecture to provide agent-level capabilities. Furthermore, from the use case scenario, it was clear that populating the various databases of the generic agent with application-specific information gave the agent domain-level problem solving capabilities. Thus, in principle, application-specific agents can be constructed directly by developers starting from the generic ZEUS agent. However, in line with our arguments in the introductory sections of facilitating rapid development of collaborative agent systems, the ZEUS toolkit has to support a more high-level agent development approach. Using such an approach, the multi-agent system developer does not need to know the details of the Agent Component Library in order to develop working agent systems. This section presents the ZEUS agent design approach, and briefly describes a visual environment that supports this approach.

4.2.1 The ZEUS agent building approach

At the highest level of abstraction, the ZEUS agent design approach requires developers to view an agent as composed of three layers: a definition layer, an organisation layer and a co-ordination layer (Figure 4). At the definition layer, the agent is viewed as an autonomous reasoning entity, i.e. in terms of its competencies, rationality model, resources, beliefs, preferences, etc. At the organisation layer it is viewed in terms of its relationships with other agents, e.g. what other agents it is aware of, what abilities it knows those other agents possess, what relationships it has with the other agents, etc. At the co-ordination layer the agent is viewed as a social entity, i.e. in terms of its co-ordination and negotiation techniques. On top of the co-ordination layer are the protocols that implement inter-agent communication. Whilst beneath the definition layer is the application programmer’s interface that enables the agent to be linked to the external programs that provide it with resources and/or implement its competencies.

Figure 4: An abstract context diagram of a generic agent
The main activities and outputs of the agent design approach are shown in Figure 5. The activities are based somewhat on our abstract high-level conceptualisation of an agent, and include (i) domain study leading to candidate agent identification and domain ontology specification, (ii) agent definition, (iii) task definition, (iv) agent co-ordination protocols specification, (v) agent organisation, and (vi) domain-specific problem solving code production. Steps 1–5 should be performed iteratively until the developer is satisfied that the final suite of agents accurately capture the details of the problem being modelled. In the proceeding paragraphs we describe each of the steps.

**Figure 5:** The stages of the ZEUS agent design approach.

**Domain Study:** At this initial stage, the developer analyses the problem domain to identify potential agents and to create an ontology of the concepts in the domain. The choice of candidate agents largely depends on the granularity at which the problem is modelled, and will be very domain- and problem-specific. However, we advocate entities with autonomous decision-making ability and operating over some sphere of responsibility as prime candidate agents. Our view is based on Jennings & Wooldridge’s [20] definition of an agent as an autonomous, social, responsive and proactive entity; and on consideration of the task-oriented nature of the ZEUS agents. The problem of devising an ontology of the domain concepts is an even more difficult issue than identifying candidate agents. However, at this stage, only the major domain concepts need to be identified and defined; lower level concepts can be deferred until the second iteration of the design process when the domain is better understood.

**Agent Definition:** Once all the candidate agents are identified, the developer next identifies the significant attributes of each agent. One metaphor we advocate for viewing an agent at this stage is as a manager in charge of a factory comprising a number of production lines (of identical machines), each of which can perform a number of different tasks. The manager has at his/her disposal a number of resources, and the production of an item may consume some of these resources, with the production process lasting for a finite time interval. The role of the manager is to produce items at the request of customers, in such a manner that he/she minimises his/her idle time and maximises his/her profit. In order to do this, the manager has to schedule customer requests on the basis of available resources, free machines and the cost and time of performing each task. Typical customer requests are of the form “produce item $u$ given $v$ by time $w$ at cost $x$”. To aid the manager’s scheduling process, he/she maintains a diary of his/her current commitments.
On the basis of the above metaphor, the Agent Definition phase involves

(i) identifying the tasks that the agent can perform — i.e. the different tasks the machines in the factory can perform and the items they produce.

(ii) identifying the number of tasks the agent can perform concurrently — i.e. the number of independent production lines in the factory.

(iii) identifying how far ahead in time the agent normally plans its activities — i.e. the maximum duration over which the factory manager books customer requests. In some scenarios, managers who can only plan ahead for shorter periods are less able to meet customer requests but may be more reactive to changing circumstances.

(iv) identifying the initial resources available to the agent — i.e. the initial resources available to the manager.

The Agent Definition phase continues until all agents have been considered, and their initial resources and tasks have been identified.

**Task Definition:** At this stage, tasks that were identified at the Agent Definition phase are defined in terms of their preconditions, effects, cost, duration, and constraints. The preconditions list the resources needed for execution of the task, while the effects list the resources that will be produced upon execution of the task. Typically, the preconditions of a task are consumed in order to produce its effects. The cost and duration are arithmetic expressions that return in some notional units, the cost of executing the task and the length of time the execution process is expected to take. Typically, expressions for cost and duration are functions of the effects and/or preconditions of the task. The constraints of a task either relate some preconditions and/or effects of the task to one another or impose some applicability restrictions on the operator.

If a task can be directly executed by a domain function, then it is identified as a primitive task; otherwise, it is identified as a summary task. For a primitive task, its precondition order and the domain function that implements the body of the task are also declared. The precondition order of a task is a partial ordering of its preconditions, constraining the sequence in which they must be achieved. This is useful in cases where the actual domain realisation of one precondition determines the acceptability of others. For example, in trip planning [27], the preconditions of a task may be that a flight is booked and transportation to the airport arranged. In such a case, it is reasonable to book the flight first, since arranging transportation to the airport requires foreknowledge of the airport from which the flight departs.

Summary tasks are useful for representational and cognitive economy (and planning efficiency). A summary task is composed of a number of other subtasks that need to be performed in some order to achieve its effects. Thus, in fact, summary tasks are mini-plans. Summary tasks are described in terms of a number of nodes and effect-precondition (producer-consumer) links between the nodes. Each node is defined in terms of its preconditions and effects, similar to primitive tasks. However, a node is simply a placeholder that can be replaced by any primitive or summary task whose preconditions and effects are supersets of the node’s. Like primitive tasks, summary tasks also have associated duration, cost and constraints; however, they lack a reference to an execution function since they cannot be executed directly.

Since the preconditions and effects of tasks must be domain ontology concepts, defining the tasks of each agent forces the developer to revisit the problem of devising an ontology for the domain concepts. However, by this point, the developer has more focus, and can now identify and define many low level ontology concepts deferred from the initial Domain Study stage.

**Agent Organisation:** At this stage, the developer identifies the acquaintances of each agent. To acquaint an agent $A$ with another agent $B$, the resources that agent $A$ believes agent $B$ can produce are specified, along with the primary organisational relationship that agent $A$ believes it shares with agent $B$. Furthermore, for each of the resources that agent $A$ believes agent $B$ can produce, agent $A$’s beliefs about the average cost that agent $B$ charges for the resource, along with the average time it takes agent $B$ to produce the resource are identified. Note that organisational relationships are not by default bi-directional; that is, although an agent $X$ might believe another agent $Y$ to be her subordinate for example, it does not necessarily imply that agent $Y$ believes agent $X$ to be her superior. Note also that one agent’s beliefs about a second agent need not be consistent with a third agent’s beliefs about the second agent.

**Agent Co-ordination:** This phase involves identifying the appropriate co-ordination protocols each agent is likely to require for social interaction with other agents when performing its designated duties. Examples of such co-ordination protocols include master-slave for delegating tasks to subordinates, contract-net for contract-
ing tasks out to peer agents, and various auction protocols for buying and/or selling resources. As the mechanics of these protocols have already been defined in the Agent Component Library, the developer only needs to decide which protocols are most applicable to an agent given its societal status and role.

**Code Generation and Task Implementation:** By this, the final stage, all the information necessary to automatically generate source code implementations for each agent should be available. The only missing ingredients are the program code that implement the body of primitive tasks. The Agent Building Software contains a Code Generator Tool that can automatically generate individual agent programs given their specification. Also, for each primitive task, it generates stub code that can be filled in by the developer with the appropriate task body. Furthermore, for user-designated agents, it generates stub code that allows external extension of the agent’s resource database.

### 4.2.2 The ZEUS Agent Generator: supporting the ZEUS agent building approach

The ZEUS Agent Generator is a 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.

The reader should note how the agent building approach and the ZEUS Agent Generator derive from the functional requirements of Section 3.2.

### 4.3 The ZEUS Utility Agents

Section 3.1 noted our desire for the toolkit to have certain utility agents that provide the infrastructure of a multi-agent system. The ZEUS suite of utility agents consists of a nameserver and a facilitator agent for 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 in-
formation 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 new 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. For a more detailed description of visualisation and debugging in ZEUS see [29].

5 USING THE ZEUS TOOLKIT TO CONSTRUCT A SIMPLE APPLICATION IN SUPPLY CHAIN PROVISIONING

This section describes a simple example of the use to the ZEUS toolkit to construct a multi-agent application. The example is in the domain of supply chain provisioning, which is often cited as a particularly appropriate application area for agents [2,26]. The hypothetical scenario involves a number of companies involved in the assembly of computer-related products. Each company in the scenario specialises in the assembly of a number of product lines, where each product may require components that need to be purchased from other companies. In the remainder of this section, we describe how a developer would apply the ZEUS agent design approach to create a working application.

**Domain Study:** The first stage in creating this application is to analyse the problem scenario. Part of this activity is the identification of candidate agents. In this case, the decision will be primarily based on existing organisational boundaries within the problem domain, (i.e. the company boundaries). The short-list of candidate agents is shown in Table 1.

<table>
<thead>
<tr>
<th>Agent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1, C2</td>
<td>Computer assembly</td>
</tr>
<tr>
<td>M1</td>
<td>Monitor and CPU assembly</td>
</tr>
<tr>
<td>M2, M3</td>
<td>Monitor assembly</td>
</tr>
<tr>
<td>U1, U2</td>
<td>CPU assembly</td>
</tr>
<tr>
<td>P1, P2</td>
<td>Printer assembly</td>
</tr>
<tr>
<td>T1, T2, T3</td>
<td>Printer parts manufacture</td>
</tr>
</tbody>
</table>

**Table 1:** Candidate agents in hypothetical supply chain provisioning scenario.
Choosing what to model as an agent enables the developer to set the level of abstraction of the solution. Our scenario illustrates this, since by creating one agent for each company, the developer has ensured all inter-agent interactions will occur at the company level.

The next activity is an ontological analysis of the domain; this will reveal the principal concepts shared by the agents. In this case, these will be the product descriptions, e.g. descriptions for items such as computers, keyboards, monitors, processing units (CPU), printers, toners, etc. This information is entered into the ZEUS Ontology Editor, which can be invoked at any time during the design process to add or amend concepts.

The remaining stages of the ZEUS agent design approach are then followed for each of the agents identified. For the remainder of this section we shall focus on the design of C1, a computer assembly agent, and P1, a printer assembly agent.

**Agent Definition:** To define an agent the developer must specify its planning capabilities, that is, the dimensions of its plan diary, the tasks it can perform and the set of resources it possesses. In the case of agent C1, it possesses, say, two production lines, and accepts customer request as far as 14 days ahead. To express a time period the developer must first determine the granularity of time the application will use. So, if the smallest unit of time to be represented is one hour, then the value of this agent’s plan diary length will be 24x14=336. Also, since C1 has two production lines, its plan diary width will be 2.

Next, the developer must identify the abilities of agent C1, i.e. the tasks it can perform. After considering all the assembly operations that this agent performs, the developer enters a single task called **AssembleComputer** into the Agent Editor. At this stage, the developer only identifies the tasks the agent can perform, leaving their specification for the **Task Definition** phase. Finally the resources owned by the agent are entered. For example, to represent the fact that the computer manufacturer has a stock of 50 UK keyboards, the developer will specify that the agent should be created with a **Keyboard** fact, with its number attribute set to 50 and its type attribute set to “UK”. The existence of **Keyboard** items and their attributes must, of course, have been previously specified through the Ontology Editor. Figure 6 is a screenshot of the ZEUS Agent Generator definition of agent C1.

In the case of agent P1, assume that from the scenario description, it has one production line (i.e. a plan diary width of 1) and schedules its activities over a 3 weeks period (i.e. a plan diary length of 24x21). Further, it can perform two tasks, **AssembleLaserPrinter** and **AssembleInkjetPrinter**, and it has a stock of inkjet and laser printer cases.
Figure 6: Definition of agent C1.
**Task Specification:** This stage involves specifying the details of each of the tasks identified during the previous stage. For example, consider the `AssembleComputer` task. In our hypothetical scenario, the assembly of a computer is a primitive task that get executed directly in the domain. The task requires a Monitor, a Keyboard, a Printer and a CPU as its inputs, whilst the effect of performing the task is the production of a Computer. Figure 7(a) illustrates the Task Definition Editor specification of this task, while Figure 7(b) illustrates constraints relating the Keyboard precondition of the task to its Computer effect. The constraints restrict the number of Keyboard items required as input to the task to be the same as the number of Computer items required as output, and the type of the Keyboard items to be the same as the `kb_type` of the Computer items. As shown on Figure 7(b), such constraints are typically embedded as the values of the attributes of the preconditions and effects of a task. The Agent Component Library supports a rich language for representing and processing constraints that includes arithmetic, if-then-else and logical constraint expressions.

**Figure 7:** (a) The `AssembleComputer` primitive task specification.
The other attributes of the primitive `AssembleComputer` task that need to be specified are its cost, duration, applicability constraints, precondition order, and the domain function that implements the task. From Figure 7(a), the cost of performing the task is specified as 1000 monetary units per computer, and the duration is 1 time unit per computer. No applicability constraints or preferred precondition order are imposed on the task. Finally, the Task Editor assumes the name of the domain function that implements a primitive task to be the same as the task’s name.

The `AssembleLaserPrinter` and `AssembleInkjetPrinter` tasks of agent P1 can be specified in a similar manner to encode the fact that the assembly of a printer requires an appropriate `PrinterCase` and either a toner or inkjet `Cartridge` depending on the type of printer.

**Agent Organisation**: This stage requires the developer to specify any pre-arranged agreements between the agents, and any beliefs one agent has about the capabilities of other agents. For instance, agent C1 may have a long-standing agreement to use agent M1 as a preferred supplier. Creating a co-worker relationship between the agents can represent this agreement. As a result, agent C1 will always attempt to obtain supplies from agent M1, and only consider other sources if agent M1 is unable to supply them. Also, agent C1’s beliefs about the capabilities of agent M1 can be represented by declaring what agent C1 believes agent M1 can produce. Figure 8 is a screenshot illustrating the Organisation Editor specification of the acquaintance relationships and beliefs of agent C1.
In the case of the acquaintance model of agent P1, assume that it owns as subsidiary (companies) the agents T1 and T2; T1 is used for the production of toner cartridges and T2 for inkjet cartridges. This relationship can be represented by declaring agent P1 as the superior of agents T1 and T2.

Note that it is not mandatory for one agent to have a relationship with every other agent in the scenario. For example, in our specification so far, agent C1 has no knowledge of the existence or capabilities of agent P1. However, if agent C1 needs to find a printer manufacturer then a suitable supplier (perhaps agent P1) can be discovered at run-time by consulting a facilitator agent.

**Agent Co-ordination:** As explained in Section 4, agents interact with one another using the social interaction protocols with which they have been equipped. Thus, for the agent C1 to contract out the production of a Monitor to agent M1, both parties must possess the contract-net co-ordination protocol. Alternatively, where one agent is subordinate to another, the master-slave delegation protocol is more appropriate. Thus, in our scenario, agent C1 can be equipped with the contract-net protocol for contracting tasks to Printer, CPU and Monitor assembly agents; and agent P1 can be equipped with both master-slave and contract-net protocols. The master-slave protocol will be used for its dealings with its subordinate agents (T1 and T2), while the contract-net protocol will be more appropriate for interactions with its peers such as agent C1.

**Code Generation:** When the previous five stages have been completed for every agent the description held by Agent Editor can be automatically converted into Java source code. The developer’s sole responsibility will be to supply the code that implements the bodies of all the primitive tasks defined in the scenario.

During this Code Generation stage, the developer can also decide to generate utility agents such as nameserver, facilitator and visualiser agents.

### 5.1 Running the Application

Figure 9 is a screenshot of three of the ZEUS visualisation tools depicting the society, task decomposition and statistics views when a goal to assemble a computer is given to agent C1. From the task decomposition view, we can see that agent C1 contracted out the supply of an appropriate Monitor to agent M1, a Printer to agent P1 and a CPU to agent U1. Given that agent C1 has a preferred supplier relationship with agent M1 who can produce both Monitors and CPUs, one might have expected the CPU contract to be given to agent M1 instead of agent U1. The reason the contract was given to agent U1 is because agent M1 produces only CPUs of a particular type, which was not the same as that specified in the goal given to agent C1.
Figure 9: A screenshot of three of the visualisation tools. In the Society Viewer (top-left) each icon represents a single agent; the envelopes shown moving between agents signify the transmission of messages. The toolbar buttons on the top left of the Society Viewer window control the position and visibility of the agent icons, while the buttons on the top right control the ‘video replay’ facilities. The Statistics Tool (middle-right) shows a histogram analysis of the type of messages being exchanged in the problem solving process for the specified goal. The Reports Tool (bottom) shows the agents responsible for the various subtasks that form the task currently being attempted by a group of agents. The colour-coded state of each task is also shown, i.e. firmly booked, running, completed, failed, etc.

5.2 Summary

The last two subsections illustrated how the ZEUS toolkit can be used to rapidly develop a working multi-agent application. The example was deliberately kept simple for expositionary reasons; however, the toolkit can also be used for more complex task-oriented applications. We have already utilised it to develop prototype applications in

- travel management [27] where a travel manager agent plans transatlantic trip itineraries on behalf of users, negotiating with other hotel, airline, taxi and train reservation agents, and communicating results to...
users via electronic mail and/or telephone messages;

- telecommunications network management, where agents collaborate to provision a virtual network link for users; and

- workflow management where agents collaborate to control the flow and processing of work items through a number of distributed and independent work processing centres. The workflow management prototype is our most advanced to date, involving the integration of ZEUS agents with a commercial workflow system. It is a ZEUS-based implementation of the system described in Judge et al. [21], and was designed to assess on the one hand whether the ZEUS toolkit achieved its main aims of facilitating the development of multi-agent systems, and the other hand whether agent-based workflow management would provide any real benefits. This work is still ongoing and the results so far are encouraging although they have highlighted a number of possible improvements to the ZEUS toolkit (which we discuss in Section 7).

We have also applied the toolkit to electronic commerce applications, where the agents are engaged in buying and selling goods. The commerce prototypes (described in [9]) were designed to test the flexibility of the Coordination Engine in representing different market-oriented co-ordination protocols such as auctions; and its ability to dynamically vary the negotiation strategy, even within the course of a single negotiation instance.

In the next section, we describe the design and implementation of the main processing components of the generic ZEUS agent.

## 6 IMPLEMENTATION OF THE AGENT COMPONENT LIBRARY

This section describes some further design and implementation details of the Agent Component library. We concentrate on some of the main processing components of the generic 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. Those not interested in implementation details may skip this section of the paper.

It is worth noting at this point the main design principle underpinning the design of the various components. The key idea was to 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 Software and processed accordingly at runtime by the agent. Furthermore, the behaviour is declaratively specified, then it can be modified dynamically, even at runtime. Therefore, the idea was to have the generic ZEUS agent function as an ‘interpreter’ of specified behaviour, so that the behaviour relevant to different agents in different domains could be defined independent of the actual implementation of the ZEUS agent. In subsequent subsections, we describe how this was achieved for the main processing components of the generic ZEUS agent.

### 6.1 Communications Management

#### 6.1.1 The Mailbox

Communications is managed in a ZEUS agent through the combined actions of its Mailbox and Message Handler. The Mailbox is responsible for receiving and dispatching messages, with a typical Mailbox maintaining two independent threads of activity — one, a reader thread, to continually listen for incoming messages and another, a writer thread, for dispatching messages. Communications is via point-to-point TCP/IP sockets, with each message communicated as a sequence of ASCII characters. On receipt, the ASCII sequence is parsed into a performative object and the object queued onto a priority FIFO incoming-message-queue. In the current implementation of ZEUS, the performative object uses the KQML agent communication language syntax, but includes some additional fields. Figure 10 illustrates a sample performative object.

---

2 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.
Figure 10: A sample performative object.

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.

6.1.2 The Message Handler

The Message Handler functions as the 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.

\[
\text{message-pattern} \rightarrow \text{action-type} \ \text{object-reference} \ \text{method-name} \quad /* \text{Object Rules} */ \\
\text{message-pattern} \rightarrow \text{action-type} \ \text{fully-qualified-graph-name} \quad /* \text{Engine Rules} */
\]

message-pattern is a partial description of a message performative that gets matched against the new 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. action-type can be set to either EXECUTE_ONCE or EXECUTE_MANY, where the rule is deleted from the Handler after execution in the case of EXECUTE_ONCE, whereas for EXECUTE_MANY it is retained after execution. The fully-qualified-graph-name is a string reference to the qualified name of one of the Co-ordination Engine graphs (described later).

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 [1]. 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.

Object rules are used in implementing short-lived and simple reactive behaviour, for example automatic responses to requests for information. Engine rules, on the other hand, are more useful 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

\[
\text{reply-message-pattern} \rightarrow \text{action-type \ object-reference \ method-name}
\]

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.

Regarding the idea of declarative specification of processing behaviour, in the design of the Message Handler outlined above, this is supported through the use of pattern-action rules. Thus, the processing behaviour of the Message Handler can be modified, even at runtime, simply by adding new processing rules or deleting existing ones.

6.2 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.

6.2.1 Problem solving behaviour representation and processing

In the final design, problem solving behaviours are represented as recursive transition network graphs, which are interpreted by a recursive finite state machine. Figure 11 depicts a code fragment defining a simple graph and its equivalent pictorial representation. As shown on the figure, 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 11, 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 11: Behaviour graph representation — simple_graph.](image)

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 node processing succeeded, while 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 11 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 be then only 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. Finally, we believe the transition network representation to be more intuitive.

**6.2.2 The default problem solving behaviour**

The generic ZEUS agent predefines a default goal-processing graph that is shown depicted in Figure 12 and Table 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 describe the behaviour of the graph with an example.
**Figure 12:** Simplified version of the default goal processing graph (the housekeeping nodes and arcs are not shown).

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

**Table 2:** Descriptions of the nodes and arcs of Figure 12.

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 12), 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 (got from the content of the message that is 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. At S2, the agent prepares to contract out the subgoals y and z. This is done 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 subgoals 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.

![Figure 13: The A4 parallel graph.](image)

The graph defined by A4 (an example is shown in Figure 13) simply serves to provide a placeholder for 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. The graph processor attempts to traverse the first viable arc from S2a to S3a in Figure 13. Thus, the choice of negotiation protocol depends first on 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 back-tracking 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 13) 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 sub-contracts, 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 of 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 requested 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 subgoals 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 representa-
tion 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 13, and S5 and S6 of Figure 12, respectively. In fact, as mentioned in Section 4, the generic ZEUS agent has some predefined negotiation protocols such as contract-net, master-slave delegation and some auction protocols. Figure 14 and Table 3 illustrate our current initiator- and respondent-side implementations of the contract-net protocol. In typical negotiation graphs, the negotiation strategy logic are 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 14 mandates a preference for co-worker agents before peer agents when contracting out tasks. The ordering of the arcs A1 and A2, where A1 precedes A2 indicates this.

![Initiator-side protocol](image1)

![Respondent-side protocol](image2)

**Figure 14:** Sample ZEUS implementation of the contract-net protocol.

<table>
<thead>
<tr>
<th>Node/Arc</th>
<th>Description/Transition Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Identify agents that can perform goal</td>
</tr>
<tr>
<td>A1</td>
<td>Select subset of agents that can perform goal and who are co-workers (check ≠ ∅)</td>
</tr>
<tr>
<td>A2</td>
<td>Select subset of agents that can perform goal and who are peers (check ≠ ∅)</td>
</tr>
<tr>
<td>S2</td>
<td>Send request for proposals to selected agent and await responses</td>
</tr>
<tr>
<td>A3</td>
<td>Check that an accept response has been received</td>
</tr>
<tr>
<td>S3</td>
<td>Done</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node/Arc</th>
<th>Description/Transition Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>S4</td>
<td>Evaluate cost Send accept message Await response</td>
</tr>
<tr>
<td>A4</td>
<td>Contract award message received</td>
</tr>
<tr>
<td>S5</td>
<td>Done</td>
</tr>
</tbody>
</table>

**Table 3:** Descriptions of the nodes and arcs of Figure 14.
6.2.3 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 described in the preceding paragraphs. 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.\(^3\)) What is particularly noteworthy is that in the current version of the Co-ordination Engine there is no explicit rationality model, that 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 [35]. Given the application domains for which we designed ZEUS, such an explicit rationality model was not deemed particularly necessary, although we plan to include one in future versions of the system.

6.3 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.

6.3.1 Plan operator representation

As described in Section 4.2.1, 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.

For planning efficiency, we also allow the preconditions and effects of operators certain qualifications (similar to variable and method 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. Further, preconditions marked as local imply the Planner can only apply the operator if it possesses the resource or it can produce the resource on its own; that is, the agent cannot delegate production of the required resource to another agent. Effect conditions can be marked as either public or private. Private effects are in fact side-effects, and constrain the Planner from selecting the operator in order to produce such effects. Thus, the Planner selects an operator to produce a desired effect only if that effect is marked public.

6.3.2 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 [17]). Thus, given a goal of the form of Figure 15(a), the Planner searches its Plan Database for an operator with a public effect that unifies (with unification bindings \(\theta\)) 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 \(\theta\), 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 (see Figure 15(b)). The idea, borrowed from job-shop scheduling, being that an agent schedules 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 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 that requested the task whether it can perform the

---

\(^3\) 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.
goal. The confirm_time specifies the latest time by which the agent that requested the task will inform the agent performing the task 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.)

```java
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 15: (a) The goal data structure. (b) A plan diary.

If an operator is successfully scheduled into the plan diary, then the Planner attempts 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 the precondition, (b) a matching unused effect (i.e. a side effect) of a previously scheduled operator can be reserved for the precondition, (c) an operator that produces the required precondition can be scheduled into the diary, and (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.

To illustrate the anchoring process, 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 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. Given a sequence of preconditions such as \([a] \rightarrow [b, e] \rightarrow [d]\), if for example, \( c \), which must precede \( d \), 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 either case, if a partial plan has been constructed, the planner returns 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.

### 6.3.3 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 16 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 \texttt{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\(^4\) the applicable operators found and then selects the highest ranked one for inclusion in the plan. If the selected operator is a summary one, then the process just described is repeated recursively on the selected operator. 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 \texttt{node\_2} and \texttt{node\_4} of Figure 16 anchors the precondition, \( e \) of \texttt{node\_4}. Finally, the Planner attempts to anchor all remaining unanchored preconditions in the expansion; e.g. the precondition, \( b \) of \texttt{node\_2} or \( a \) of \texttt{node\_1}.

---

\(^4\) The ranking criteria is now by summary operators first, then by cost, and finally by duration.
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.

6.3.4 Managing resource reservation conflicts

It is perhaps worth remarking about the mechanism used to ensure the coherence of produced and supplied items when subgoals of the form produce α given β are passed around in 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.

6.3.5 Plan execution and exception handling

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, 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 failure 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 such a case, 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 either by (i) allocating a new resource to it – if one can be found, (ii) creating a new plan to produce the required resource, (iii) creating a subgoal 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. Tables 4 and 5 present an example scenario illustrating some of the mechanisms involved in the exception handling process (not included in the scenario are situations where existing contracts need to be cancelled).

<table>
<thead>
<tr>
<th>Task Database</th>
<th>Agent1</th>
<th>Agent2</th>
<th>Agent3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task Database</td>
<td>{b}→Task1/10→{a}</td>
<td>{c}→Task4/40→{b}</td>
<td>{b}→Task4/40→{a}</td>
</tr>
<tr>
<td></td>
<td>{c}→Task2/20→{b}</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>{d}→Task3/30→{c}</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>{x}→Task6/60→{c}</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>{y}→Task7/70→{b}</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>{z}→Task8/80→{a}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resource Database</td>
<td>{c,d,x,y,z}</td>
<td>{c}</td>
<td>{b}</td>
</tr>
<tr>
<td>Key</td>
<td>{Preconditions}→TaskID/Cost→{effects}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Initial states of the three agents involved in the exception handling scenario described in Table 5.

<table>
<thead>
<tr>
<th>Action/Comments</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>achieve Agent1 a</td>
<td><img src="image1.png" alt="Diagram" /></td>
</tr>
<tr>
<td>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.</td>
<td><img src="image2.png" alt="Diagram" /></td>
</tr>
<tr>
<td>delete Agent1 c</td>
<td><img src="image3.png" alt="Diagram" /></td>
</tr>
<tr>
<td>Deletion of the database resource, c, leads to a new operator (Task3) to anchor the precondition of Task2.</td>
<td><img src="image4.png" alt="Diagram" /></td>
</tr>
<tr>
<td>delete Agent1 d</td>
<td><img src="image5.png" alt="Diagram" /></td>
</tr>
<tr>
<td>Deletion of the resource, d, means Task3 loses its anchor. Hence, the planner backtracks, and tries an alternative mechanism for anchoring the precondition, c, of Task2.</td>
<td><img src="image6.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

5 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.
delete Agent1 \( x \)

Again, more backtracking.

---

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 tried, 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 \( a \).

---

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, \( a \).
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 5: An exception handling scenario.

6.4 Integrating ZEUS Agents with External Programs

The preceding sections mentioned in passing the primary interfaces between ZEUS agents and external programs. These include the domain functions in primitive plan operator specifications, and user-defined Coordination 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 the size and functionality of the Resource Database. 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. Furthermore, the agent 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. Figure 17 illustrates a code fragment of an external Java program that utilises the ZeusExternal interface and the agent internal event model to monitor the addition of facts into an agent’s Resource Database. Whenever a fact of type Car is added into the database, the program immediately deletes it.

---

6 The current version of the system does not even attempt to manage any internal inconsistencies that might result from such actions.
public class Control implements ZeusExternal {
    protected ZeusAgent agent = null;

    public void exec(ZeusAgent agent) {
        this.agent = agent;

        agent.addFactMonitor(
            new FactAdapter() {
                public void factAddedEvent(FactEvent event) {
                    String type = event.getFact().getType();
                    if (type.equals("Car"))
                        agent.getResourceDb().deleteAll(type);
                }
            },
            FactEvent.ADD_MASK
        );
    }
}

Figure 17: Integrating ZEUS agents with external systems – using the ZeusExternal interface and the agent internal event model. The addFactMonitor() method registers a new event monitor with the agent, asking for notification whenever a new Fact is added into the agent’s Resource Database. The monitor is implemented using the inner FactAdapter class.

7 DISCUSSIONS AND RELATED OTHER WORK

7.1 Discussion

In the previous sections we outlined the philosophy and aims of the ZEUS project, and described the toolkit that was developed to realise those aims. To a large extent, we can claim that the toolkit has met both its functional and non-functional requirements. Particularly, it has facilitated the creation of working multi-agent systems by users with only basic competence in agent technology. This, we believe, is a direct result of our separation of the agent-level issues from the domain-level ones, by insisting on a declarative specification of behaviour that is interpreted by the generic ZEUS agent. However, we have yet to conclusively assess our requirement that the toolkit should support large-scale software engineering of distributed systems. To assess this requirement, significant evaluation in terms of mainstream industrial use is required. We have begun such an evaluation in the agent-based workflow management domain, where we utilise agents to control a commercial workflow management system. The preliminary results of this exercise have been encouraging, but highlighted a number of areas where the ZEUS system could be improved.

The first major area for future work concerns the development of methodologies for analysing domains in order to identify candidate agents and characterise their interactions. This is particularly so in highly unstructured domains, where perhaps an agent-based approach would provide the greatest benefits. Research on role-based modelling of agents may provide some pointers towards a solution.

Other concerns particular to the ZEUS system include the need for an explicit rationality model to control agents’ behaviour. While incorporating one into a ZEUS agent is not particularly difficult (see Section 6.2.3), the real problem lies in devising some mediating representation [5] through which relatively naïve users can program the rationality model. Similar concerns can be raised about user-defined co-ordination behaviour. The recursive transition network approach to representing co-ordination behaviour appears intuitive to expert users, but our experience suggests that non-expert users face difficulty using it, especially when they need to consider the multiple potential behaviours of two parties in a negotiation dialogue. (However, in the case of defining co-ordination behaviours, this problem is alleviated significantly by our provision of a library of common negotiation protocols). A final area highlighting the need for better mediating representations concerns the incorporation of a mechanism for reactive problem solving in ZEUS agent. Again, a classical condition-action representation and processing of low level behaviours could be incorporated into ZEUS agents; but the problem, as always, lies in how users can declaratively program the conflict resolution strategies that are invoked when reactive data-driven behaviour and deliberative goal-driven behaviour interfere with one another. Furthermore, such conflict resolution strategies would depend on the rationality model in place.

In future work, we plan to investigate the above issues, as well as incorporate some learning and more advanced scheduling mechanisms into the ZEUS Agent Component Library.
7.2 Related Work

There are several notable attempts at building test-beds for multi-agent systems that are relevant to ZEUS, but these are aimed primarily to support the implementation of ideas so that they can be assessed in a meaningful context. Such test-beds include DVMT [24], MACE [12] and ARCHON [42]. On the other hand, ZEUS provides a development environment.

There are other attempts at building agent development environments for agents of differing types and complexities. The ADEPT Project [33] produced an agent architecture (or framework) for business processes. The RETSINA architecture [38] is a distributed collection of goal-oriented software agents that co-operate asynchronously to perform information retrieval and integration for supporting various decision-making tasks. Work at the University of Toronto on an agent building shell [2,3] is also related to ZEUS. This work describes an agent building shell “that provides several reusable layers of languages and services for building agent systems: co-ordination and communication languages, description logic based knowledge management, co-operative information distribution, organisation modelling and conflict management”. This work is still much in progress and has not yet resulted in a practical toolkit embodiment as ZEUS or a coherent architecture like RETSINA, with much effort having been expended on issues such as description logics, non-monotonic truth maintenance in agent-based systems, and on the extension of KQML to derive the language COOL. The Toronto work clearly recognises and begins addressing some important issues that are identified in Ndumu & Nwana [28]. We borrowed from some of this work, although the conversation rules in their COOL language take a rather top-down approach to managing conversations between agents, while our co-ordination graph specification is a more bottom-up design. Jackal [34] is an agent communication infrastructure to support agent-to-agent communication using the KQML ACL. Jackal also implements various conversation policies like in COOL.

DESIRE [7] provides a compositional development method and architecture for multi-agent systems. It focuses on the architecture, and attempts to provide generic compositional “building blocks” for specifying agent architectures, like in ZEUS. However, DESIRE does not prescribe or specify issues such as protocols, for example for inter-agent collaboration; nor does it support truly distributed agent systems. Also, DESIRE does not offer similar visualisation and debugging facilities as in ZEUS. DESIRE has been used to develop several prototype multi-agent applications.

The Open Agent Architecture (OAA) [25] is a truly open architecture for developing distributed multi-agent systems. Its key distinguishing feature is its powerful facilitator that co-ordinates all the agents in the set-up, and which is also a planner. The facilitator can receive tasks from agents, decompose them and award them to other agents in the set-up. All agents must communicate via the facilitator, which can become an obvious bottleneck. OAA also employs a custom primitive agent communication language. It is designed to minimise the effort involved in creating new agents and “wrapping” legacy applications. Several OAA-based systems exist [25]. OAA principally differs from ZEUS in that co-ordination in ZEUS agents is decentralised, while in OAA it is centralised around facilitators. Also, OAA does not generate code as in ZEUS, and does not offer similar visualisation and debugging facilities.

The dMARS architecture [19,35], developed at the Australian Artificial Intelligence Institute, has its conceptual underpinnings in the belief-desire-intention (BDI) model [6, 15], and it provides an environment for developing distributed multi-agent systems. Thus, dMARS agents attempt to operationalise the BDI reasoning framework. Like ZEUS, the primary reasoning mechanism in dMARS agents is planning, with intentions operationalised as plans. (In dMARS however, planning is more reactive than in the current version of ZEUS). In dMARS there is also less emphasis on co-ordination of multi-agent activity. Thus, there is no explicit co-ordination engine (or equivalent), and it is unclear how general purpose co-ordination protocols and strategies can be shared transparently among dMARS agents. Perhaps, the key non-functional difference between dMARS and ZEUS is that of abstraction. We believe dMARS requires agent developers to work at a much lower level of abstraction than that required by the ZEUS Agent Building Environment.

JAFMAS (Java-based Agent Framework for Multi-Agent Systems) is primarily concerned with providing communication and co-ordination support for agent system developers [8]. Co-ordination is supported through use of conversation classes that agents utilise to manage their interactions with other agents during problem solving. The conversation classes implement rule-based automata models, similar in spirit to the way co-ordination behaviour is managed in ZEUS. Thus, a conversation class might implement the equivalent of the contract net co-ordination protocol, but specific to an agent and a given context. JAFMAS agents use broadcast communication to perform the services of the ZEUS Facilitator agent, i.e. identifying agents with a particular capability.

KAoS is an ambitious project to devise an industrial-strength open agent architecture. One of its main distinc-
tions from similar work is its explicit concern with security issues. Thus, in collaboration with Sun Microsystems, the K AoS design team are attempting to build security models that separate policy from mechanism. This way, agents may also negotiate about what security models to use in addition to negotiating about services. K AoS agents also provide conversation management support (as in JAFMAS) and an optional planner (as in ZEUS). K AoS, like ZEUS, also explicitly tries to raise the level of abstraction at which agent designers need to work to create agents. Thus, the K AoS agent design tool-kit supports mediating high-level representations for describing conversations and plans, and also intends to support one for describing security models [4,5]. As we discussed in the previous subsection, we believe this issue of mediating representation is extremely significant if agent-based systems are to find widespread industrial use.

JAFIMA (Java Framework for Intelligent and Mobile Agents) is a layered framework for the design of agent systems. It is designed and documented using patterns, and is also explicitly concerned with how the agents integrate with other system software. JAFIMA is perhaps one of the most comprehensive agent design frameworks, but in contrast to the toolkit approach of say ZEUS or dMARS, JAFIMA is primarily targeted at developers wanting to develop agents from scratch. So whilst JAFIMA attempts to define a complete development methodology, it does not provide pre-built agent-level components for use by developers.

8 CONCLUSIONS

A project as complex as building a generic and customisable toolkit for constructing collaborative agents truly forces its developers to confront and synthesise many disparate bodies of work. We have found this very rewarding and we certainly encourage such research which synthesises existing work, but which also offers much scope for more meaningful ‘invention of new wheels’. In carrying out this project, we have borrowed unashamedly from a wide and diverse literature including agent communication languages, distributed object technologies, co-ordination/co-operation/negotiation literature, rational agent theories, visual programming, planning & scheduling, visualisation, methodological issues, specification of ontologies, automatic code generation, HCI design, etc. In the same vein, we believe we have pushed the state-of-the-art in these broad areas:

- The production of a generic, customisable and extensible toolkit which facilitates and speeds up the development of complex agent applications. We are still to prove that it can support industrial-strength applications like dMARS;
- The visualisation/debugging of complex distributed agent-based systems;
- The derivation and implementation of a co-ordination framework which can support a host of protocols in the co-ordination literature. This enables ZEUS to be able to support a true marketplace where different agents are using one or more different co-ordination strategies concurrently.

In conclusion, this paper has described the rationale, philosophy, underlying methodology, design and implementation of an advanced toolkit for constructing distributed agent applications. We hope it provides a valuable contribution to this new and exciting area of research.

ACKNOWLEDGEMENTS

Barry Crabtree, Stuart Soltysiak and Nader Azarmi all contributed to useful discussions that shaped the ideas embodied in the ZEUS toolkit. This work was funded by BT Laboratories.

REFERENCES


