Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Document
D-96-06

Working Notes of the KI'96 Workshop on Agent-Oriented Programming and Distributed Systems

Klaus Fischer (Ed.)

August 1996

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 20 80
67608 Kaiserslautern, FRG
Tel.: + 49 (631) 205-3211
Fax: + 49 (631) 205-3210

Stuhlsatzenhausweg 3
66123 Saarbrücken, FRG
Tel.: + 49 (681) 302-5252
Fax: + 49 (681) 302-5341

Deutsches Forschungszentrum für Künstliche Intelligenz

The German Research Center for Artificial Intelligence (Deutsches Forschungszentrum für Künstliche Intelligenz, DFKI) with sites in Kaiserslautern and Saarbrücken is a non-profit organization which was founded in 1988. The shareholder companies are Atlas Elektronik, Daimler-Benz, Fraunhofer Gesellschaft, GMD, IBM, Insiders, Mannesmann-Kienzle, Sema Group, Siemens and Siemens-Nixdorf. Research projects conducted at the DFKI are funded by the German Ministry of Education, Science, Research and Technology, by the shareholder companies, or by other industrial contracts.

The DFKI conducts application-oriented basic research in the field of artificial intelligence and other related subfields of computer science. The overall goal is to construct systems with technical knowledge and common sense which - by using AI methods - implement a problem solution for a selected application area. Currently, there are the following research areas at the DFKI:

- ☐ Intelligent Engineering Systems
- ☐ Intelligent User Interfaces
- ☐ Computer Linguistics
- ☐ Programming Systems
- ☐ Deduction and Multiagent Systems
- ☐ Document Analysis and Office Automation.

The DFKI strives at making its research results available to the scientific community. There exist many contacts to domestic and foreign research institutions, both in academy and industry. The DFKI hosts technology transfer workshops for shareholders and other interested groups in order to inform about the current state of research.

From its beginning, the DFKI has provided an attractive working environment for AI researchers from Germany and from all over the world. The goal is to have a staff of about 100 researchers at the end of the building-up phase.

Dr. Dr. D. Ruland
Director

Working Notes of the KI'96 Workshop on Agent-Oriented Programming and Distributed Systems

Klaus Fischer (Ed.)

DFKI-D-96-06

This work has been supported by a grant from The Federal Ministry of Education, Science, Research, and Technology (FKZ ITW-95 004).

ISSN 0946-0098

Working Notes of the KI'96 Workshop on Agent-Oriented Programming and Distributed Systems

Klaus Fischer (Ed.)

August 27, 1996

Preface

Distributed AI (DAI) and Multiagent Systems (MAS) are nowadays not only integrated into the programme of any national and international AI conference but in many application domains the developed techniques are already applied in practical applications. Intelligent agents raise requirements to interaction abilities that go far beyond the pure functional interoperability. When pursuing local or global goals, agents must be able to coordinate their activities, exchange knowledge, and resolve conflicts.

Recently new technologies were developed which are likely to influence DAI and MAS research. This is especially true with industrial standards like ODP, OMG, and CORBA with respect to distributed object-oriented systems from which at least some of the MAS development tools have emerged. New programming languages for `mobile agents' (e.g., Java, Telescript etc.) stimulate current research. The goal is now to develop agent architectures which meet the requirements of these standards and innovations with respect to openness, tractability, and security aspects.

We require tools for agent-oriented software development and it is necessary to develop methodologies to validate MAS.

The papers in these working notes present subjects like:

- usefulness of object-oriented analysis and design methodologies for MAS
- coordination of active agents in open systems
- communication concepts for MAS
- agent-oriented perspectives to ODB, OMG, and CORBA
- usefulness of Java to describe and implement agents in a MAS

The overall goal of the workshop is to stimulate the discussion on how the new developments can be effectively integrated into the current research work.

Klaus Fischer

Content

Models and Methodology for Agent-Oriented Analysis and Design

Birgit Burmeister
Daimler-Benz AG, Research Systems Technology, Alt-Moabit 96a, 10559 Berlin, Germany
bur@DBresearch-berlin.de

Abstract

Agent-oriented techniques are likely to be the next significant breakthrough in software development process. They provide a uniform approach throughout the analysis, design and implementation phases in the development life cycle.

Agent-oriented techniques are a natural extension to object-oriented techniques, but while there is a whole plethora of analysis and design methods in the object-oriented paradigm, very little work has been reported on design and analysis methods in the agent-oriented community.

After surveying and examining a number of well-known object-oriented design and analysis methods, we argue that none of these methods, provide the adequate model for the design and analysis of multi-agent systems. Therefore, we propose a new agent-specific methodology that is based on and builds upon object-oriented methods. We identify three major models that need to be build during the development of multi-agent applications and describe the process of building these models.

1 Introduction

Agent-oriented programming or in more general terms agent-oriented techniques (AOT) provide a new approach that aims at supporting the whole software development process. Analysis, design, and implementation are done in a simple and natural way, at a level of abstraction more adequate to the problem to be solved. The goal of AOT is to handle all phases with a single, uniform concept, namely that of agents.

During the analysis phase the acting entities of the problem domain are identified and modelled as agents. Agents and their actions (or behavior) are refined and specified in the design phase. Finally, at the implementation phase, agents are programmed with the aid of an agent-oriented programming language or using a multi-agent development environment.

AOT are a natural extension to object-oriented techniques (OOT), that are also aimed to support all phases of software development in a general and uniform way. Agents can be seen as active objects. The differences between objects and agents, as stated by [Sho90], are (i) the structuring of the internal state of an agent by mental notions like beliefs, goals, intentions, and the like, and (ii) characterization of messages by message types and the structuring of messages into protocols. As a result of these conceptual differences, agent-oriented systems need to be analyzed, designed and implemented differently.

While there is a plethora of analysis and design methods in the area of OOT, very little has been done for analysis and design in the agent-oriented community. This paper is aimed as a con-

tribution to this area. By looking at some well-known object-oriented analysis and design methods we highlight the inadequateness of these methods for modelling multi-agent systems. Nevertheless inspired by OOT we introduce a set of models and outline the process of building these models for the development of multi-agent systems (MAS).

In section 2 we give a short survey of the general concepts in OO analysis and investigate whether these concepts can be carried over to AO analysis. Based on this in section 3 we will present our models and a methodology that builds upon and uses some notations used in OO methods. Finally, we give an overview of the related work in section 4 and conclude with a summary and an outlook in section 5.

2 Object-Oriented Analysis

There are quite a number of approaches to object-oriented analysis (and design)[1] such as the Object Modelling Technique [RBP+91], Responsibility Driven Design [WWW90], Object-Oriented Software Engineering (Objectory) [JCJ+92], Object-Oriented Design [Boo91], and the Fusion Method [CAB+94]. Abstracting away from their finer grained differences, common to most of these approaches one could identify the following step-wise procedure:

1. The first step in the majority of methods is identifying the objects and classes in the system.
2. After the objects/classes are identified, the static relationships among them are specified. These relationships are inheritance, aggregation or more general association relationships.
3. Then the dynamic relationships, (i.e. events and messages) are specified. As for the dynamics, two aspects have to be considered: (i) 'dynamics in the large', that is, the messages exchanged between objects that build up the system behavior, and (ii) 'dynamics in the small', that is, the internal flow of states and events/messages within one object (sometimes called object life cycle).
4. Finally the internal structure of the objects is described, i.e. the attributes are defined and the operations (methods) of the object are described.

The result of these steps can be seen as three sub-models of a complete model of the system to be built. These models are named *basic*, *static* and *dynamic* models in [Bal94].

- The basic model contains the objects/classes and their attributes and operations.
- The static model contains the structure of the system as described by the relationships among objects/classes as inheritance, aggregation or more general associations and grouping into subsystems.
- The dynamic model contains messages and interaction diagrams as well as objects' life cycles and the specification of object operations.

1. Since there is a fluent borderline between analysis and design in OOT, we will use the term "OO analysis" instead of "OO analysis and design" throughout this paper.

After a thorough investigation of OO analysis and design techniques we found that they are not directly applicable to the development of multi-agent systems. This is basically due to their conceptual differences between objects and agents.

1. Agents have a more complex behavior and structure than objects, and in this respect they are more comparable to subsystems in some OO methodologies, (e.g., [WWW90]). Their internal structure differs from objects in that agents have a more complex underlying functional architecture such as the belief-desire-intention (BDI) architecture [RG92]. In this respect they are on a higher level of abstraction than objects.

2. Unlike objects whose internal states are defined in terms of some arbitrary attributes, the internal states of agents are defined in terms of some mental notions, like beliefs, plans and goals, which distinctly characterize the agent.

3. In contrast to objects that are rather passive entities, agents are active. Objects immediately become active through messages and in this respect they are benevolent, whereas agents act on their own behalf by following their goals, and can decide whether they act and respond to events and the messages received from other agents.

4. Agents' behaviors are described as scripts or plans by some graphical means that resemble state transition diagrams used to describe object life cycles in some OO techniques, (e.g., [RBP+91]). Agents' plans can be directly implemented using an agent programming language or an appropriate tool or environment such as DASEDIS [Bur93] or dMARS [Kin93]. Specifically using such tools, implementing an agent is mainly specifying its plans.

5. The communication of objects only looks at single messages. In AO not only messages are characterised by message types, agents dialogues with respect to specific contexts are prestructured into cooperation protocols like the ones proposed in our earlier work [BHS93].

The Responsibility Driven Design (RDD) method [WWW90] for the design of object-oriented systems comes closest to the concepts in AOT. In this method identifying and specifying responsibilities are the dominant starting point. A class inheritance hierarchy is build up using the responsibilities identified. Collaborations among objects are defined by *contracts* and *protocols* (i.e., the formal specification of method calls). Finally the system is structured by subsystems consisting of closely collaborating objects. But as was stated earlier, due to the conceptual differences between agents and object, OO methods are not immediately applicable to the design and analysis of agent-oriented systems.

This result has led us into studying and proposing a more adequate approach for agent-oriented analysis and design. The approach draws upon OO methods and uses some of the notations commonly used in many OO techniques. The next section describes this approach.

3 Agent-Oriented Analysis: Models & Methodology

Similar to the three models to be specified and built during OO analysis (i.e., basic model, static model and dynamic model), we divide the results of an agent-oriented analysis into three submodels: the *agent model*, the *organizational model* and the *cooperation model.*

- The agent model contains agents and their internal structure, described in terms of mental notions such as goals, plans and beliefs or whatever structure deems to be appropriate to an agent architecture. This model resembles the basic model of OO methods.

- The organizational model specifies the relationships among agents and agent types. These are on one hand inheritance relations (among agents and agent types, and agent types and sub- or supertypes), and on the other hand relationships among agents based on their roles in organizations. These organizations can be means for structuring a complex system into subsystems (as done in some OO techniques) or can be used to model real organizations. This model is in some respect similar to the static model, but since roles can change over time it is not genuinely static model.

- The cooperation model describes the interaction or more specifically the cooperation among agents. This model only contains the 'dynamics in the large' part of the OO dynamic model. The 'dynamics in the small' part, (i.e. the description of agent behavior), is part of the agent model.

Although these models are not strictly disjoined they can be developed separately. In the remaining part of this section we describe each of these models and outline how they may be developed. In contrast to most OO techniques however, we do not prescribe which model has to be developed first. In our experience in building MAS for real world applications the appropriate choice is ruled by the nature of the application being developed.

As a general guideline, it seems useful to start with an informal description of the scenario to be modelled before actually developing the models. This is also the first step in some of the OO techniques or is even a pre-requisite to applying the method. After this very first description one of the three models is developed, starting with the model most appropriate in the application considered.

To illustrate the methodology and how one may develop these models we will sketch an example from one of the applications we are currently developing. As part of a project investigating the possible implications of data highways and high performance computing on road traffic, we are developing a simulation system for widely decentralized and self-organized allocation of parking places. This system should allow for the simulation of different scenarios to experiment with different strategies and organizational forms and will be implemented as a multi-agent system.

The scenario to be simulated is briefly described as follows: The scenario consists of a road network, where cars move from their start node to an end node. At their end node, it is always preferred to park as close to the destination as possible. Parking places must be allocated dynami-

cally by cars negotiating among themselves (for all the parking places in the vicinity of the end node), or by consulting a node manager. In this example we will only consider the latter. A node manager is in charge of the parking places at a specified node.

3.1 Agent Model

The agent model contains the agents and their internal structure. It is built following the steps described below:

1. Identify agents and their environment.
   In many cases identifying agents and their environment is rather intuitive. Agents are the live and "active entities" in the system in that they can change their own states and their action can affect their environment, whereas the environment consists of passive elements whose state only changes by agents actions. It is quite usual for the collection of agents to grow during the analysis process. For instance, at later stages one may include agents that realize the system internal purposes. As with the RDD method, it is a good practice to create a CRC-card[2] for each agent. In the RDD method, CRC-cards are used to document classes. To document agents the CRC-card is enhanced by predefined 'attributes' like beliefs, motivations and plans identified in the remaining steps below. Other 'attributes' such as those related to the cooperation partners, must be analyzed during building the other models and added to the CRC-card afterwards.

2. For each agent define its motivations.
   Motivations (e.g., interests, preferences, responsibilities, long-term goals and so on) play an important role in the way an agent makes its decisions, and in this respect, they characterize the agent. Their role is especially important when an agent has more than one way of acting in a given situation. For instance, long-term goals lead to the selection of more concrete goals, subgoals and plans. And motivations are used to control the decision making at branching points within plans. These issues are also added to each agent's CRC-card.

3. Define the behavior of each agent.
   The behavior of agents is defined in some scheme called plans. Plans describe the sequence of actions an agent may carry out to fulfil certain motivations or react to the occurrence of certain events. Therefore plans must be related to the specific motivation or events that make them applicable and added to the CRC-cards.
   Plans can be specified in a graphical notation similar to state transition diagrams. These specifications are refined successively during the process of analysis and design towards a complete specification. As mentioned earlier, the complete specification can then be implemented directly when using an appropriate tool.

4. Define knowledge and belief.
   Finally the knowledge and belief an agent uses to execute its plans is defined and added to the CRC-card.

2. CRC stands for 'classes–responsibilities–collaborations'.

At the end of these steps there is a collection of agents with the specification of their motivations and plans in some sort of operational scheme, and the type of knowledge and belief they require to execute the plans.

In the example sketched out earlier, cars and node managers are the active entities and therefore can be modelled as agents. We introduce two types of agents, namely *car-agent* and *manager-agent*. The road network itself is not active and therefore not modelled as an agent, but is simulated as the environment of the agents.
The long-term goal of a car-agent is to drive from its start to its end node. The car-agents' preference is to park as near to its destination as possible. Node managers administer parking places at their node. Their task is to optimally satisfy the parking wishes of car-agents and cooperate with other manager-agents.
The major behaviors (plans) that can be associated to a car-agent are driving on the road network, announcing a desired parking location, and negotiating about a parking place. To be able to carry out these plans, a car-agent must have knowledge about its start and end nodes and the route to take. In case of the fully decentralized scenario, a car-agent must also have information about the availability of parking places at its destination. The manager-agents handle the parking wishes that were announced by car-agents, and administer the parking places that they are in charge of. Therefore, they need knowledge about the capacity and allocation of their parking places. Due to space limitations the successive refinement of these plans down to the operational schemes is not presented here.

3.2 Organizational Model

In this model, (roles of) agents are classified and related to each other.

1. Identify roles in the scenario
   The roles and responsibilities in the scenario are identified. If a real organization is to be modelled in the multi-agent system the roles appearing in that organization should be stated. Roles are then mapped to agents and agent types, where they may influence the motivations and therefore behavior of the agents.

2. Build an inheritance hierarchy
   Different roles or agents can be classified according to their knowledge and belief, motivations and behavior. Agents having the same beliefs, the same goals and the same behavior are instances of an agent type. As in OO techniques the common 'attributes' can be defined in the agent type and inherited by the agents. Abstract super-types can be introduced that do not have instances. The inheritance hierarchy can be represented in a notation used by OO methods, (e.g., the object diagrams of the Object Modelling technique (OMT)).

3. Structure roles into organizations
   Organizations can be used to structure a complex system into smaller subsystems. Subsystems are then parts where roles that are interacting more closely are put together. (This is also the way subsystems, clusters etc. are used in some OO techniques.)
   Another way of using organizations within the system is to model real organizations, where

roles have certain relationships to each other. The organization can also be represented by notations from OO methods, (e.g., the object model in OMT).

The organizational model gives an overview of the connections among agents and agent types, the roles and organization of agents.

The organizational model for the example scenario is quite simple. There are only two roles/ agent types, namely cars and node managers. The cars are driving and have wishes to park, these wishes can be handled by node managers. No inheritance relationships are present.

3.3 Cooperation Model

The cooperation model describes the interaction among the agents. It is built using the following steps.

1. Identify cooperations and cooperation partners
   In this step, it is stated which goal of an agent must be fulfilled by cooperation with other agents. This can be necessary to share resources, to synchronize actions or to coordinate behavior. The cooperation partners of an agent and the reason for cooperation must be entered into the agent's CRC-card.

2. Identify message types
   To carry out a cooperation we assume that agents must be able to communicate. For a meaningful communication we consider using message types. A set of predefined standard message types should be specified as proposed by KQML [FWW+94] or in our previous work [BHS93].

The result of steps 1 and 2 can be noted in a sort of collaboration or interaction graph, as used in OO techniques.

3. Define cooperation protocols
   From the interaction graphs (and sometimes from an intuitive semantics of message types) cooperations protocols can be derived. A cooperation protocol defines the possible flow of messages among cooperating agents. Where it is convenient, cooperation protocols can be built from a few basic protocols. In earlier work [BHS93], [Had96] we have identified some basic protocols for *Informing, Querying, Proposing* etc. Application specific protocols should be noted. As with behaviors (i.e., scripts or plans), protocols can also be represented in a graphical notation that is operational.

Therefore, the cooperation model states the kinds of interaction/cooperation going on among the agents and the contents exchanged by messages.

There are two types of cooperations in the parking management scenario: a) cooperation between car-agents and node managers who respectively announce and handle parking wishes, and b) cooperation between neighboring node managers who negotiate about parking places when one of the managers runs short of parking places in the area of its control. For cooperation of type (a) the corresponding message types are: *announce* a parking wish (car-agent), *confirm* the

allocation of a parking place (manager), and *propose* a different parking place (manager). For cooperation of type (b) the corresponding message types are: *inform* about allocation of parking places, *request* for a parking place, *offer* a parking place, and *reject* a request. We have designed a set of generic protocols that can handle some of these cooperation forms. Examples of these protocols can be found in [BHS93].

The three models described above together constitute the complete model of the system to be realised. This model can now be implemented directly with an appropriate tool: agent types are defined by their knowledge and beliefs, their motivations (e.g., goals) and behaviors in the form of scripts/plans, and their cooperation protocols.

4 Related Work

So far very little work has been reported in the area on the systematic and methodological analysis and design approaches for building multi-agent systems. Some of these works are briefly discussed here.

- An approach that was inspired by the KADS-model is mainly concerned with knowledge acquisition aspects [OG92]. No hints are given on how to come to an agent-oriented model of the domain.
- Some general criteria for a modular system design from OO techniques as defined in [Mey88] were transferred to multi-agent system design by [OW92].
- The project CONSENSUS [CON92] proposed a very high-level two step approach. The first step being a 'normal' requirements analysis as done for any software system. The second step is the design of a multi-agent system. However, again no instructions were given as how this may be done.
- The approach described in [Dor93] discusses the decisions that have to be made when developing a multi-agent system, and in this respect is closer to our approach. The decisions concern the definition of the agents, the description of their beliefs and capabilities and finally the specification of generic interactions and communication contents. However, no guidelines are provided as how appropriate structures and models may be developed to facilitate making "good" decisions.
- Recent work of Kinny and his colleagues [KGR96] is very similar to the work presented in this paper. Starting from a specific OO technique, namely OMT, they propose a methodology and modelling approach for multi-agent systems. They distinguish between an external and an internal model. The external model consists of an agent model (our organizational model) and an interaction model (our cooperation model), and is independent of a chosen agent architecture. The internal model (our agent model) is specific to the BDI agent architecture and describes the agents by a plan model, a goal model and a belief model. The internal model is a specialization of our agent model and is quite elaborated. On the other hand their interaction model is not as worked out as our cooperation model.
- In the work of Rosenschein et al., [RZ94], the main focus is on the cooperation and interaction aspects of multi-agent systems. They describe the definition of a negotiation process (as a special form of cooperation), as a three step task. Namely defining the space of

possible deals, the negotiation process (as seen from the outside) and the negotiation strategy (for each agent). These instructions can be seen as a special form of building the cooperation model.

5 Summary and Outlook

We have presented an approach for a systematic development of multi-agent systems. We have introduced three distinct models and some model building instructions.

The motivation for our work was the need for a systematic approach for developing multi-agent systems for real world applications and the fact that very little effort has been made in the AO community on this subject.

Since AOT are a specialization of OOT and there exist many OO analysis and design techniques, we examined whether they could be used for the analysis and design of multi-agent systems. We found that none of the methods considered can be used immediately, without further modifications, mainly because there are important conceptual differences between objects and agents and consequently the resulting system models. However, OO techniques provided a good starting point for our work, especially the RDD method, which is conceptually closest to AO modelling.

Our approach consists of three distinct models that are built during the process of analysis and design: The agent model, consisting of the agents and their internal structure defined in terms of knowledge and belief, motivations and behavior; the organizational model, describing the structure of the system and modelling organizations (real or artificial) agents are working in; and the cooperation model, containing the interactions among the agents, i.e., describing who interacts with whom, for what purpose, using which message types and protocols.

For each model we gave some instructions as to how the model may be built. These three models together describe the multi-agent system to be realised.

Most of the reported work on this topic is related to the agent model, so this is the most detailed model. For BDI agents in particular, we found the methodology given in [KGR96] to be a recommended alternative.

Our previous work on cooperation protocols influenced the cooperation model. Defining appropriate cooperation protocols and the way scripts or plans are used to program the agent's behavior is an important issue in the development of multi-agent systems and needs to be elaborated further.

The least elaborated model is the organizational model. The impact of a role an agent takes within an organization on its behavior has not been investigated thoroughly. Furthermore so far there is no way of directly implementing organizations by corresponding language constructs. One way would be to look at organizations as special agents, that contain special joint plans the member of the organization can use (or inherit) as Shoham proposed in his first discussion of agent-oriented programming [Sho90].

Our approach is currently being tested in a number of real world applications. This will give us more insights into the development process for multi-agent systems and thus will help to refine and improve the models and methodology.

Acknowledgement

I would like to thank the members of the 'agents' team at Daimler–Benz Research for their input and fruitful discussions. Special thanks to Afsaneh Haddadi for helping to make this a readable paper.

References

[Bal94] H. Balzert: *Methoden der objekt-orientierten Systemanalyse,* Angewandte Informatik 14, BI-Wissenschaftsverlag, Mannheim, 1994. (in german)

[Boo91] G. Booch: *Object-Oriented Design with Applications*, Benjamin/Cummings, Redwood City, CA, etc.,1991.

[Bur93] B. Burmeister: "DASEDIS – Eine Entwicklungsumgebung zum Agenten-Orientierten Programmieren" in: H.J. Müller (ed.): *Verteilte Künstliche Intelligenz*, BI–Wissenschaftsverlag, Mannheim, 1993. (in german)

[BHS93] B. Burmeister, A. Haddadi, K. Sundermeyer,: "Generic Configurable Cooperation Protocols for Multi-Agent Systems" in: C. Castelfranchi, J.-P. Müller (eds.): *From Reaction to Cognition. Proc. MAAMAW'93*, LNAI 957, Springer, Berlin, etc., published 1995.

[CON92] "Methodology Report (D13)", British Aerospace, Durham University, Cambridge Consultants, 1992.

[CAB+ 94] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, P. Jeremaes: *Object-Oriented Development. The Fusion Method*, Prentice Hall, Englewood Cliffs, 1994.

[Dor93] J. Doran: "Using DAI Software Testbeds" in: *Proceedings CKBS'93*, DAKE Centre, Keele, 1993.

[FWW+94] T. Finin, J. Weber, G. Wiederhold, M. Genesereth, R. Fritzson, D. McKay, J. McGuire, R. Pelavin, S. Shapiro, C. Beck: "Specification of the KQML Agent-Communication Language" (Draft), The DARPA Knowledge Sharing Initiative, External Interfaces Working Group, 1994.

[Had96] A. Haddadi: *Communication and Cooperation in Agent Systems: A Pragmatic Theory,* LNAI 1056, Springer, Berlin, etc., 1996.

[JCJ+92] I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard: *Object-Oriented Software Engineering. A Use Case Driven Approach,* ACM Press/Addison-Wesley, 1992.

[Kin93] D. Kinny: "The Distributed Multi-Agent Reasoning System Architecture and Language Specification", Australian Artificial Intelligence Institute, Melbourne, 1993.

[KGR96] D. Kinny, M. Georgeff, A. Rao: "A Methodology and Modelling Technique for Systems of BDI-Agents" in: W. van der Velde, J. Perram (eds.): *Agents Breaking Away. Proc. MAAMAW'96*, LNAI 1038, Springer, Berlin, etc.,1996.

[Mey88] B. Meyer: *Object-oriented Software Construction*, Prentice Hall, Englewood Cliffs, 1988.

[OG92] A. Ovalle, C. Garbay: "Towards a Methodology for Multi-Agent System Design" in: *Proceedings Expert Systems 92*, Cambridge, 1992.

[OW92] G.M.P. O'Hare, M.J.Wooldridge: "A Software Engineering Perspective on Multi-Agent System Design: Experience in the Development of MADE" in: N. Avouris, L. Gasser (eds.): *Distributed Artificial Intelligence - Theory and Praxis*, Kluwer Academic, Dordrecht, etc., 1992.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorenson: *Object-Oriented Modelling and Design*, Prentice Hall, Englewood Cliffs, 1991.

[RG92] A. S. Rao, M. P. Georgeff: "An Abstract Architecture for Rational Agents" in B. Nebel, C. Rich, W. Swartout (eds.): *Proc. International Conference on Principles of Knowledge Representation and Reasoning (KR-92)*, Morgan Kaufmann, San Mateo, 1992.

[RZ94] J. S. Rosenschein, G. Zlotkin: *Rules of Encounter. Designing Conventions for Automated Negotioation among Computers,* MIT Press, Cambridge MA, London, 1994.

[Sho90] Y. Shoham: "Agent Oriented Programming", Stanford University Technical Report STAN-CS-90-1335, Stanford, 1990.

[WWW90] R. Wirfs-Brock, B. Wilkerson, L. Wiener: *Designing Object-Oriented Software*, Prentice-Hall, Englewood Cliffs, 1990.

Communication Concepts in Multiagent Systems

Klaus Fischer[1]
DFKI GmbH, Stuhlsatzenhausweg 3, 66123 Saarbrücken
kuf@dfki.uni-sb.de

Abstract

Basic communication concepts are overall important for the deign of agents in a multiagent context. Although some researchers investigate in how far agents are able to solve a given problem in a multiagent world without the explicit use of communication, in most cases researchers assume that the agents have the ability to communicate as a basic requisite. In this paper I compare the basic communication concepts of MAGSY and MAI^{2}L with those of JAVA. Main goal of the paper is to extract the common core of these three approaches for the implementation of basic communication techniques and starting from this core shed some light on which path should be selected to further extend the concepts.

1 Introduction

Up to now is the formula $DAI = MAS + DPS$ — where DAI: Distributed AI, MAS: Multiagent Systems, and DPS: Distributed Problem Solving — the best characterization of the research area DAI. When we investigate DAI research topics, we do this in most cases by looking at a MAS which is designed to solve a problem in a specific application domain. Although some researchers investigate in how far agents are able to solve such a problem without the explicit use of communication, in most cases the researchers assume that the agents have the ability to communicate as a basic requisite.

Basic concepts to describe communication among agents in a MAS are therefore a crucial part of the agent description. However, there is no common agreement on how a standard to describe the basic communication concepts should look like. In this paper we compare the basic communication concepts of MAGSY and MAI^{2}L — selected as representatives for MAS development environments — with those of JAVA. JAVA was selected because it attracted significant attention in the context of distributed programming in the INTERNET and has been also suggested to be adequate to describe agents in a MAS context.

2 MAGSY

Originally MAGSY [FW92, Fis93] was designed to allow the implementation of a MAS which is organised as a distributed blackboard system in an easy manner. In MAGSY each agent is a knowledge-based system, which uses a forward-chaining rule interpreter as basic inference machine. These concepts have been extended in the projects AKA-MOD and COMMA-MAPS at DFKI GmbH in Saarbrcken and resulted in the agent architecture INTERRAP.

[1]This work has been supported by a grant from The Federal Ministry of Education, Science, Research, and Technology (FKZ ITW-95 004)

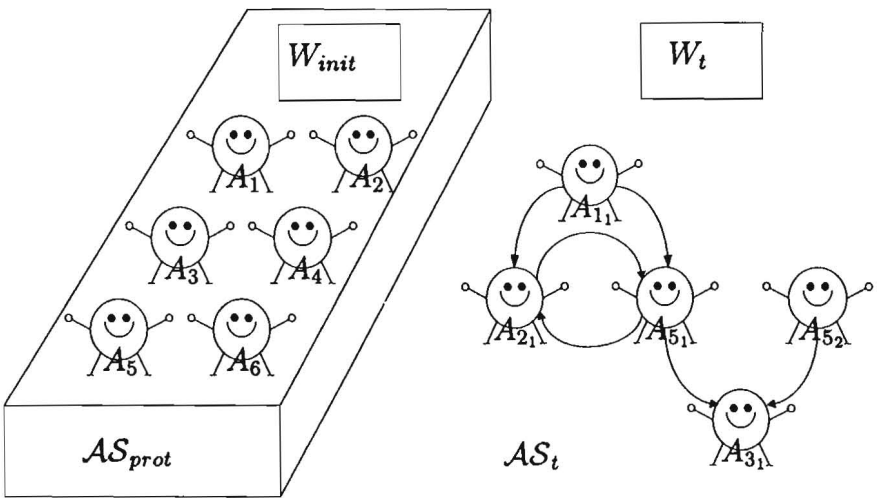

Figure 1: Static specification $\mathcal{AS}_{prot}$ of the solution for a problem and dynamic execution $\mathcal{AS}_t$

In MAGSY an agent is defined by a triple $(\mathcal{F}, \mathcal{R}, \mathcal{T})$, where

$\mathcal{F}$ is a set of facts which represent the local knowledge of the agent.

$\mathcal{R}$ is a set of rules which define the strategies for the general behaviour of the agent.

$\mathcal{T}$ is a set of services which are provided by the agent.

For a given agent A, $A_{\mathcal{F}}$, $A_{\mathcal{R}}$ and $A_{\mathcal{T}}$ denote the respective sets. Agent A may receive messages which change the set of facts or activate a service of the agent. The basic communication layer of MAGSY provides therefore the basic speech acts inform and request, where the semantics of the message is fixed by the knowledge representation in the receiving agent. Each service of an agent consists of a set of rules which become active when the specific service has been activated by an incoming message. The execution of a service by an agent A may change the set of facts $A_{\mathcal{F}}$ and the set of rules $A_{\mathcal{R}}$. Therefore, the agents are learning when they execute their services, which can also mean that rules and facts can be forgotten, i.e. are removed. It is also possible that the execution of a service by agent A creates a number of new agents. The requests for services which are received by an agent are treated according to a specific strategy. Only one service can be active at a time. The other requests are queued and activated when the current service is finished. A priority is assigned to each service. A service s may be interrupted by a service i if a message arrives which activates i and i has a higher priority than s. If i is finished, s is reactivated. This process may happen recursively. At any point in time the service with highest priority is active. If more than one request for this priority class is present, the request belonging to the oldest message (that is the earliest one received) will be executed.

The knowledge of the agents is represented in an object-oriented knowledge representation scheme. Objects are the basic elements which build a knowledge base. Objects which have the same structure are grouped together and make up classes. The description of a class is a prototype object which specifies how the elements (we call these elements instances or facts) look like. The properties of these instances are described by attributes. The classes are structured in

a hierarchy in which attributes are inherited from the upper, more general, classes to the lower, more specific ones.

To describe the solution of a given problem, a set of cooperating agents is specified. The static description of the solution of the problem is given by the prototypical agent system

$$\mathcal{AS}_{prot} := (\mathcal{A}_{prot}, KB_{init}),$$

where

$\mathcal{A}_{prot}$ is the set of prototypical agents, the agents which may arise dynamically in the system. These agents are the potentially available problem solvers. Several instances of a specific prototypical agent can be created.

KB_{init} is a prototypical agent with a fixed set of services. It is designed to provide an active, global knowledge base and provides agent directory services.

The process of problem solving starts with the initial agent system

$$\mathcal{AS}_{init} = ((A_{1_1}, \ldots, A_{n_1}), KB_{init}),$$

where $A_{1_1}, \ldots, A_{n_1} \in \mathcal{A}_{prot}$. The computation goes on while the agents perform requested services and call on services of other agents. In doing so the agents are cooperating by sending messages to each other. New agents may be created and killed again later on. A message may contain the identification of an agent, thus any desired structure of communication can be created dynamically. The contents of the global knowledge base can be accessed by all agents. Thus, the global knowledge base may be changed by the agents in any desired manner. For example, an agent can make its own identification known to all of the other agents by writing it into the global knowledge base.

In the example of figure 1 the system started with $\mathcal{AS}_{init} = ((A_{1_1}, A_{5_1}), KB_{init})$. A_{1_1} creates A_{2_1} and A_{5_2} and A_{5_1} creates A_{3_1}. At time t, A_{1_1} told A_{5_2} the identification of A_{2_1} and A_{5_2} told A_{2_1} its own identification. The identification of A_{3_1} came known to A_{5_2} because A_{3_1} stored its identification in the global knowledge base and A_{5_2} extracted this information from the global knowledge base. In this example there is no other way for A_{5_2} to get the identification of A_{3_1}.

3 MAI^{2}L

MAI^{2}L [Kol95] is an agent-oriented programming language and has been developed in the DFKI projects KIK-TEAMWARE and COMMA-PLAT. One of the basic requirements for MAI^{2}L was that it should provide a platform for the development of MAS in an industrial context where efficiency in execution and in use of resources is of overall importance.

Agents in MAI^{2}L are objects and agent live as threads in a multi-threaded virtual machine. Communication among agents is possible within a virtual machine as well as among agents in different virtual machines distributed on serveral computers. The agents knowledge base is organized by association and contexts. Roughly spoken the knowledge base of an agent is a collection of dictionaries which map identifiers to values. Contexts (i.e. dictionaries) have to be opened to become active and values for identifiers defined in a dictionary which was newly opended overwrite values for these identifiers which were defined in dictionaries opend beforehand.

There are a few predefined agent classes — agent directory service (ADS) and monitor — which are needed for debugging and the maintainance of multiagent systems. Agents in MAI^{2}L can be multi-threaded in itself. However, there are no further concepts which suggest a structure for multi-threaded agents.

Activities MAI^{2}L agents have to reason about can be specified by plans. As the agents themselves the plans are defined by an object. On the one hand, these objects contain the procedure which specify the activities to be executed for the plan. On the other hand, they provide information which allow the agent to reason about the plan. However, there is currently no basic planning mechanism included in the MAI^{2}L system. Therefore, a planning mechanism has to be built on top of MAI^{2}L. The representation of plans in MAI^{2}L allows to specify multiagent plans by labeling the activities with the characters which have to execute the activity. The following plan provides a multiagent plan for a car and a car park to synchronize their activities to allow the car to drive into the car park.

```
proc enter () {
  car: drive_to_entrance()
  car: request_entrance()
  car_park: open_barrier()
  car: drive_to_space()
}
```

The car and the car park have to open the appropriate context — providing the information which car and which car park are actually involved in the cooperation — to be able to execute the plan.

Communication among agents is done via a set of cooperation primitives. A cooperation primitive consists of a type, which specifies the intention behind the communication and a cooperation object. When an agent communicates the first time with another agent, its communicator asks its ADS about information on the network address and process number of the recepient. The communicator creates a new instance of the class *acquaintanace* which stores this information.

In the acquaintanance object all cooperation primitives are defined as methods. It represents the send-target on the sender side and does the neccessary network communication to the actual recepient. The actual message contains additional information that is added automatically by the communicator of the sender:

- a unique id to identify the ongoing cooperation and its context
- the character name and the agent name of the sender and the recepient
- the name of the cooperation method, if the method is sent within a fixed protocol
- the type of the primitive indicating the intention behind the message.

This information must be available in the cooperative context of the initiator of a cooperation method, the one that sends the first message, before the cooperation is started.

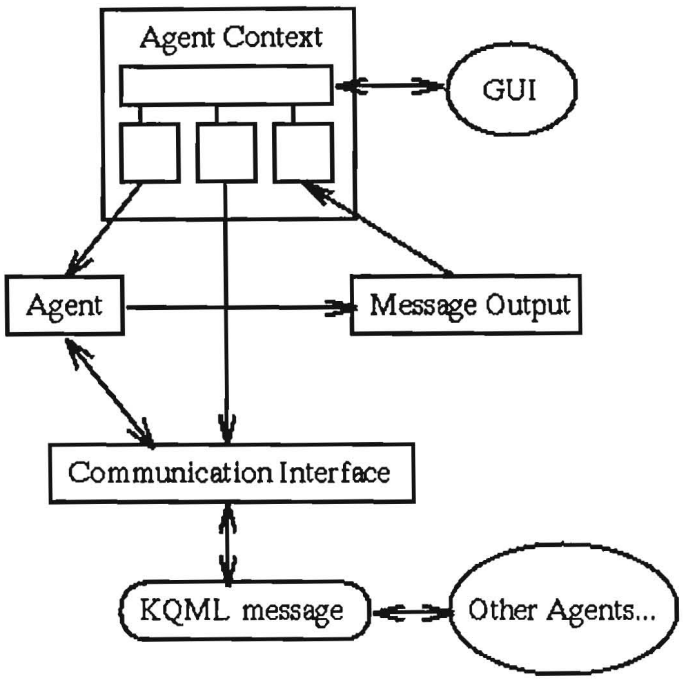

Figure 2: The architecture of an agent in the Java Agent Template (JAT).

4 JAVA

JAVA [AG96, K"u96] has recently attracted attention in the context of INTERNET applications. It has been introduced as an INTERNET language and has been widely used in browsers for the world wide web (WWW). More recent developments aime at the use of JAVA as a language to describe agents in a MAS context. Rob Frost's JAVA agent template (JAT) for example is an implementation of a set of JAVA classes which provide means to describe autonomous agents. In this section we give a brief outline of JAT so that we are able to compare it to the MAGSY and MAI^{2}L approach.

An agent in JAT is implemented as a set of basic objects as it is shown in Figure 2.

Agent Context: The agent context provides an executable container for the agent and its associated communication interface. The agent context is represented by the AgentContext interface which specifies a set of abstract methods which must be implemented by any object which servers as an agent context. In the present distribution, the AgentContext interface is implemented by the classes AgentFrame and ANSFrame which provide agent contexts with GUIs. Each of these classes can be executed as stand alone applications or as applets, via the classes AgentApplet and ANSApplet.

Agent: The agent object represents the primary functional element in the architecture. An agent basically represents a black box which asynchronously accepts and outputs KQML messages via a communication interface. An agent's knowledge and its body functions (i.e., the functions which are designed to solve specific problems in an application do-

main) are represented by resouce objects. Standard resources include addresses, languages, ontologies and classes. An Agent communicates with the agent context via a MessageOutput object.

Communication Interface: The communication interface object provides the low level mechanism for reliable transmission of KQML messages. In the present distribution, communication is done using a socket-based interface (class SocketInterface) which allows to receive and to transmit messages in a multi-threaded messsage.

Message Output: Communication between the agent object and its context is done using a message output object. This object allows the agent object to output both system and KQML messages to the user interface and execute certain context methods.

JAT provides a set of java classes as a basic toolbox to implement JAVA-based autonomous agents. Basic classes are:

Agent.class: Provides the basic agent control loop. In this control loop the agent is able to send and receive KQML messages. Furthermore, the agent is able to access knowledge and functions provided by the *resource* classes.

ANS.class: Is a sub-class of Agent.class. It provides methods to store and retrieve addresses of agents. This class is used to implement agent directory services which give agents the ability to contact other agents they originally did not know.

CommInterface.class: Is an extension of the JAVA SocketInterface.class which implements the port socket communication. It provides an interface for the agent to the mechanisms which physically transmit the message across the network. JAVA threads are created on receiving and on sending messages.

MessageOutput.class: Provides the interface between the agent and the user interface. It passes on system as well as KQML messages.

NetworkClassLoader: Is an extension of ClassLoader.class and provides mechanisms to load classes dynamically across the network. It needs the URL of the class as a parameter. In JAT it is used to dynamically load the knowledge and the functions of the agent.

There are two classes AgentApplet.class and ANSApplet.class to create an ordinary or an ANS agent, respectively. When an AgentApplet.class (ANSApplet.class) object is created, it automatically creates an object of AgentFrame.class (ANSFrame.class). These two classes open a window which provides the user interface to the agent (ANS). Furthermore, a CommunicationInterface object is created. Panels are seperate set of classes which allow the design of the user interface (e.g. ComposeMessagePanel.class, LoadResourcePanel.class etc.).

Additionally, there is a set of classes called resource classes which describe the objects an agent manipulates in its problem solving process. There are five resource classes:

Addresses: Describes adresses of agents specified by hostname and port number.

Classes: Describes the knowledge and the domain procedures and functions for the agents. It is imported by the Agent.class (or sub-classes, respectively).

Languages: Defines the syntax of the strings that can be sent in a message. A language object parses a string and returns a message object. An example for such a language is KQML.

Ontologies: Defines the semantics of message contents. An example is AgentOntology.class which supplies resources by an ANS, which defines ask-resource: a request for a resource of which only the name is known and tell-resource: an announcement of a resource by an agent. The method interpret-message takes a language object as input and interprets it. Up to now this results in immediate calls of methods which provide the domain procedures and functions of the agent.

KQMLmessages: Describes the structure of KQML messages, e.g., the performative tell:

```
:sender address-of-sender-agent
:receiver address-of-receiver-agent
:language KQML
:ontology AgentOntology
:content (tell ...)
```

The design of an agent is done by describing the domain functions, procedures, and the knowledge, and structure them into classes. To describe the messages an agent is able to process, a set of KQML performatives is selected and the semantics for the message contents is fixed. Furthermore, the system engineer has to provide methods which actually produce the message strings. The basic agent control loop is always inherited form Agent.class. The system engineer extends the basic control loop by introducing test for different message types and by linking the incoming messages to methods representing the domain functions and procdures. Finally, the system engineer has to define a set of panels to describe the user interface of the agent.

5 Comparison of the Cooperation and the Communication Concepts of MAGSY, MAI^{2}L, and JAVA

From the three systems described above the JAT seems to be the most basic system. It purely concentrates on the communication aspects of the agents. Though JAT agents communicate in terms of KQML messages, internally the agents do not have a sophisticated knowledge representation and reasoning mechanism. However, these mechanisms can of course be implemented on top of the basic JAT concepts. The basic ideas for the design of the communication interface are therefore the most interesting parts in JAT.

On a first glance MAI^{2}L seems to be a quite low-level language, too. However, it reveals its expressiveness when communication protocols are descibed. The ability to alow to write procedures which can be executed by several agents representing different characters makes it possible to describe communication and cooperation protocols in a compact manner. The bad thing about this choice is that these concepts can be used only in an agent which is actually able to interprete the MAI^{2}L language. There is no strict classification of communication primitives into performatives with a predefined semantics. Conceptually MAI^{2}L incorporates highlevel reasoning concepts like planning based on the event calculus. However, in [Kol95] there is no description of how this could be done in an efficient manner.

The basic communication concepts of MAGSY provide the performatives *inform* and *request*. Each agent has an inference procedure built into its kernel. In the original MAGSY system this reasoning procdure was purely in a forward reasoning style. Whereas this paradigm is especially adequate to describe reactive behaviour, it does not give to much support for the implementation of rational agents which have to do deliberative planning to find out which action they should actually take next to reach their goals. To come around these problems, we developed in the AKA-MOD and COMMA-MAPS project the INTERRAP agent architecture. The INTERRAP agent model integrates reactive, deliberative, and cooperative behaviour in a three-layered architecture [Mül96]. However, the languages and MAS development platforms presented in this paper are designed for the implementation of highlevel agent models like INTERRAP. The real question to ask here is, how much of the functionality of the agent model should be integrated into the implementation language and how much should be implemented on top of the implementation language. What we actually would need is a modular implementation platform, where the basic functionality any meaningful agent in a MAS needs is directly integrated into the implementation language, and functionality needed only for specific application domains can be loaded on demand.

As a common core of communication concepts of the three systems presented in this paper, we can see that an ADS is an important concept in a MAS development environment. It would be highly desireable to have a standardisized interface to ADSs. When we compare MAS development to distributed object-oriented programming ADS servers play the same role as ORBs. However, there is nothing around like the CORBA standard for ADSs. CORBA itself is not an adequate technology for MAS implmentation because it lacks the support of asynchronous communication.

All of the MAS development tools presented in this paper support asynchronous communication. However, all of these systems have their own basic communication concepts. What we need is a basic communication language which is independent from any specific implementation language. A key requirement is therefore the definition of a small set of agent communication languages and protocols (ACLPs). These languages should offer abstractions which are at the appropriate level and should not be tied to a particular transport mechanism or set of lower level architecture assumption. KQML [] is a step into the right directsion. However, the problem with KQML is that there is no content language which is widely accepted, and without content language KQML is only of limited use.

6 Conclusion

This paper presented three development platforms MAGSY, MAI^{2}L, and JAT which were especially designed to support an easy implementation of multiagent systems. When we compare the basic communication concepts used in these systems we realize that agent directory services (ADSs) and the ability to communicate asynchronously are crucial for agents in a multiagent context. However, there is no standard around like the CORBA technology is in the distributed object-oriented system community to implement this basic functionality.

Furthermore, we can see that there is agreement that communication should be based on speech acts. What we need here is the definition of a small set of agent communication languages and protocols (ACLPs). These languages should offer abstractions which are at the appropriate level and should not be tied to a particular transport mechanism or set of lower level

architecture assumption. KQML [] is a step into the right directsion. However, the problem with KQML is that there is no content language which is widely accepted, and without content language KQML is only of limited use.

References

[AG96] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[Fis93] K. Fischer. The Rule-based Multi-Agent System MAGSY. In *Proceedings of the CKBS'92 Workshop*. Keele University, 1993.

[FW92] K. Fischer and H. M. Windisch. MAGSY- Ein regelbasiertes Multi-Agentensystem. In H. J. Müller, editor, *KI1/92, Themenheft Verteilte KI*. FBO-Verlag, 1992.

[Kol95] M. Kolb. A cooperation language. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 233–238, San Francisco, CA, June 1995.

[K"u96] Ralf K"uhnel. *Die Java-Fibel*. Addison-Wesley, 1996.

[Mül96] J. P. Müller. *An Architecture for Dynamically Interacting Agents*. PhD thesis, Universität des Saarlandes, Saarbrücken, 1996.

Requirements for Enabling Intelligent Network Applications

Afsaneh Haddadi
Daimler-Benz AG, Forschung und Technik,
Alt-Moabit 96a, 10589 Berlin
afsaneh@DBresearch-berlin.de

Abstract

Since the advent of world wide web, internet has been growing in popularity not only in academia and research, but also commerce and industry. A vast array of network services is growing around internet, at the same time, telecommunications industry is rapidly advancing, wire-less/mobile communication is reaching world-wide, and 'telematic' devices are improving and becoming more accessible. Providing these services on a network, and enabling ready access to them, demand advanced but intuitive tools that not only support establishing connections and communication, but also aid in monitoring, controlling and coordinating interactions, and managing, securing and safeguarding information. This paper discusses some of the requirements for developing intelligent network applications of the furture.

1 Introduction

Since the advent of world wide web, internet has been growing in popularity not only in academia and research, but also commerce and industry. A vast array of network services is growing around internet, at the same time, telecommunications industry is rapidly advancing, wire-less/mobile communication is reaching world-wide, and 'telematic' devices are improving and becoming more accessible. All these have contributed to give distributed computing a new meaning.

Providing these services on a network, and enabling ready access to them, demand advanced but intuitive tools that not only support establishing connections and communication, but also aid in monitoring, controlling and coordinating interactions, and managing, securing and safeguarding information. In many cases the information sources are digitally available as pure (or marked-up) text, or can be accessed via databased, knowledge-based or any other application program, for example, digital libraries, yellow pages, road maps, daily city entertainment programmes, real-time traffic information and so on.

This paper promotes the development of a range of tools and a unified framework to enable "*agentification*" of application programs (of both server and client programs). Agentification of a program means equipping the program with a set of capabilities such that it can provide or access services autonomously, intelligibly and flexibly. The resulting program is then called an agent system. The tools that enable agentification of programs must provide intuitive and convenient means to:

- specify and establish addressing, connection and communication,
- enable import and export of executable program code from one node to another,
- design and implement methods of interactions and possible dialogues in those methods,

- integrate a meta level reasoning layer that would enable the agent to autonomously reason about and coordinate its interactions with other agents, and manage issues related to authentication, authorisation and in general safeguarding of information.

Some of the issues stated above are no longer research topics, and for some of the above issues various solutions have been suggested. For example there are communication protocols (the OSI reference model, and Internet Protocols) for connecting, monitoring and controlling communication; programming languages such as Java [7], [5] and Telescript [13] in support of mobile/remote program execution; various *coordination protocols* [?] (e.g., negotiation, contracting, bargaining and collaborative tasks) for establishing dialogues and coordinating interactions; and practical models of agent cognition component (such as the *'belief, desire, intention'* (BDI) architecture)[8] for meta-level, real-time reasoning in dynamic environments.

Apart from the fact that much improvement to these tools and techniques is yet to be expected, there are no unified frameworks that allow such diverse capabilities to be integrated into a program systematically. With the aim of arriving at a unified and integrated framework for developing true agent systems, this paper will discuss the above requirements in some detail, and cite some of the more advanced activities in this direction.

This article will not address any of the issues related to addressing, 'lower level' communication and communciation protocols, inter-object communication standards (such as CORBA) and object-oriented languages based on these standards, authentication, authorisation, and finally security issues related to client and server programs. These topics are extensively discussed in the other articles in this book (workshop). Instead, the paper will concentrate on the two major issues specific to "*agents*", namely, (i) issues concenrning mobility and navigation, and (ii) issues concerning 'high-level' communication, inter-agent dialogues, cooperation, and in general agent interactions. These topics will be respectively discussed in section 2 and section 3. The paper will conclude with a summary and an outlook.

2 Mobile Agent Languages and Development Tool Kits

Mobile agent technology draws upon and integrates many areas in computing and telecommunications, to enable realisation of applications over computer networks, that could not be perceived as possible, or in the best case immensely complicated. One could trace back the evolution of this technology to the data communication between computers, and the notion of *remote procedure calls (RPC)* [12]- the organising principle in computer communication networks. Then came the *remote programming* [6] concept by which one could not only call procedures in another machine, but also supply the procedures that receiving computer must execute. The central concept behind mobile agent technology is in fact the remote programming paradigm, with the aim of enabling navigation of program code (procedures or objects) through a network of inter-connected computers such as internet or intranet.

With the advances in telecommunications in mobile/wireless communication and wider availability of ever more versatile 'telematic devices', in the recent years the idea of mobile computation has been attracting larger interest. Obviously to realise sophisticated applications for this purpose, mobile agent technology plays a very important role.

General Magic's Telescript technology [13] was the first commercial product that has fully realised the concept of mobile agent technology. In Telescript the basic concept of remote programming is advanced to enable transporting of *executing* procedures [3], that is, an object or

a procedure can be launched at any time after it has already started execution. Such objects are called *agents* based on Telescript's view of conceptualising whole network infrastructure and networkinging interactions by drawing parallels to human society and the interactions among human. In such a metaphor, agents are representatives of users who undertake tasks and interact with other representatives (or agents) in order to achieve their tasks. But since these agents are armed with specific capabilities allowing them to travel around the network, being knowledgeable of the rules, communication infrastructure and tasks related to navigating and using resources on a network, they are referred to as *mobile agents.*

To enable agents travel and run on remote machines, the nodes on an agent's travelling route must have a Telescript *engine* running on them. Telescript introduces the concept of *cloud* which is a network of Telescript engines through which agents can navigate and access information. A Telescript engine is an environment that enables sending, receiving and executing agents. In Telescript it is possible to develop agents that communicate locally on one machine, or agents that reside on different machines and communicate with one another in distance. Agents must *meet* in a *place* (a designated address) and interact with each other in that place. Agents can also travel through several nodes and return back with the data they gathered on their trip.

Telescript provides an object-oriented language (*High Telescript*) to program and develop agents. This language must be compiled by a Telescript compiler into a script-like language (*Low Telescript*) which can then be interpreted by Telescript Engine. Before transmission, an agent code is converted to wireline encoding (bag of bits) which upon arrival is translated back into Low Telescript by the Telescript engine on the receiving side.

Telescript has beed uniquely designed to enable mobile agent technology and it is especially targeted for some variety of mobile computing applications. Applications can be developed in C or C++ and only the mobile agent part need to be developed in Telescript.

Currently many activities have been reported on the efforts incorporating mobile agent technology in other languages- Java language [7] being favoured by most. Java is a full-fledged language for software development. Like Telescript, Java is also object-oriented, but although it provides a library of classes for networking and developing *applets*[1], Java itself does not directly support the concept of mobile agents. However, as was mentioned earlier there are many activities that in one way or another are aimed at providing for mobile agent technology (or some subset of it). Among these are *CyberAgent* (already a commercial product) [14], *Java Agent Template* [16], *Kona Agent* [20], *Mole* [15], *Agent Builder*[18], *Aglets* [19], and Java-To-Go [17].

Most of these languages are still in their research phase and as a result, at this stage, it is hard to evaluate their strength in terms of how well they incorporate the concept of mobile agent technology. For example, some do not support the mobility of agents (e.g., Java Agent Template), some do not support agents on different sites to directly communicate with one another (e.g., Cyber Agent), some fail to undertake the necessary measures to maintain the platform independent nature of Java (e.g., Mole), and so on.

Up to this date, the concepts behind Telescript are by far the most advanced, incorporating the concept of mobile agent technology in a clean and clear fashion. Many, especially those favouring Java, object to Telescript based on the following argument: (1) Telescript is platform specific, (2) classes have to be defined and known both on the client and server platforms, and

[1]Applets are programs (objects) that can be loaded by clients from a server and run on the client machine. Applets may also be developed to run on the client machine when loaded, and provide a user interface to some application program running on the server.

(3) Telescript is not as readily available as Java (i.e., it is not a shareware). The first two objections stem from the initial stages of Telescript, where Magic Cap platform provided the only client platform that used Telescript. With General Magic's new product, *Tabriz Agentware* [21], this is no longer an issue since Tabriz enables client agents (being developed in Java, Telescript or any other language) to connect and communicate with Telescript agents (services). Of course, the objection here would be the fact that one should in addition install Tabriz on the client platform, which is also not a shareware. The second problem arises from the fact that Magic Cap is incapable of sending classes out to servers, and consequently an agent's class and all its superclasses must be defined both on the server and the client platform. This may still be the case, which is an important drawback for Magic Cap users, but it is not a shortcoming of Telescript. The third objection that unlike Java Development Toolkit[7] Telescript is not freely available, is a valid argument against Telescript as a full-fledged programming language. It is hard to predict how the market will develop, but as it stands now, Java is not a mobile agent development language, and it is questionable if any Java-based agent product would also afford to be free of charge in the future. Cyber Agent [14] for instance is a good example.

These arguments aside, one of the most important criteria Telescript must be credited for, is that it considers not only the programming and execution requirements of mobile agents, but also the whole infrastructure needed to govern and support the realisation of real world applications through this technology. This includes measures ranging from issues concerning *activeness*, *persistence*, and *permission* for access, duration and use of resources, to issues concerning the whole organisation of the network services (considered at various levels of abstraction) and directories that aid reaching and accessing those services. The only other approach that has also considered the issues concerning the organisation of services on the network is reported in [10], and is still at its research phase.

This section predominantly discussed the mobility aspect of agents, while the next section will concentrate on *agents* as they have been traditional studied in AI and multi-agent systems, and discuss in what way the result of these studies are useful for intelligent networking applications.

3 High-Level Agent Communication and Interactions

With the growing success in mobile agent technology, once more real challenges will be converging on problems requiring more efficient and more intelligent agents. Agents will be expected to undertake more sophisticated tasks, for which they may need to collaborate and negotiate with other agents. To do this they may need to communicate more intelligibly and exhibit sufficient flexibility to build up their dialogues dynamically and interactively. This has been one of the critical lines of research in the field of Multi-Agent Systems (MAS). Typically agents in MAS are computational programs inhabiting dynamic and unpredictable environments, as is the case with agents navigating on the net. They are equipped with sensoric and effectoric capabilities, and some internal processing component that relate the sensoric events to appropriate actions effecting the environment. Agents have sufficient degree of decision-making autonomy and can interact with other agents by explicit communication.

The ability to perform "meaningful" dialogues is particularly crucial when we have an open system where (heterogeneous) agents "enter and leave" the system with little or no information about each other's capabilities, such as how to communicate with one another, which resources

to share, how to coordinate their activities, whether and how to cooperate with one another and so on.

By high-level communication we refer to the applications layer (the seventh layer) in the OSI reference model for communication. This layer itself can be viewed as consisting of various components. For instance standards for object to object communication (e.g., CORBA [11]), standards for message format/syntax in agent to agent communication (e.g., KQML [4], and standards or protocols for agent to agent dialogues and interactions (e.g., *coordination protocols* [2]). CORBA for instance specifies standards for object to object communication for a virtual object-oriented language. KQML on the other hand defines a message format standard to aid interpretation of messages in a more cognitive level (as opposed to the physical layer such as message encription for efficient and secure message delivery). Messages are characterised by *performatives* which denote communicative actions like some networking operations (such as *pipe* and *recruit*) or (some virtual) database operations (such as *delete*, and insert a fact). In KQML, the structure of dialogues are implicit, in other words, the choices of response of the receiver of a message must be implicitly encoded by the application developer. For heterogeneous agents which are likely to be designed, and developed by different authorities, this could prove problematic. This is because there is no representation or specification that explicitly stated what a permitted response to a message with a specific performativeis.

In our earlier works [2], [9], we suggested that since agents of a dialogue may be heterogeneous, and developed and owned by different authorities, in design of protocols as general interaction means, we should be able to fulfill the following requirements:

- A representation that is intuitive in terms of sketching various states of dialogue, and independent of application domains and their requirements.
- Protocols must be at a level of abstraction separate from the internal components of individual agents and their reasoning mechanisms. But their representation must explicitly specify the interface to the internal components of the agents.
- Ideally, it must be possible to design and develop protocols obeying modular design conventions, enabling rapid prototyping for analysis and experimentation.

The ideas presented in these publications are now more thoroughly worked out and will be reported in our forth-coming publication. The essential ingredient of this work is that protocols should not only explicitly represent the temporal relationship between messages and the type of messages that may be communicated at various stages in the dialogue for a specific context (e.g., negotiation, bargaining, task delegation, contract net etc.), but also specify the constraints that must be imposed and applied at various stages of a dialogue. Returning to the simple querying protocol, the protocol must specify the constraints imposed on the type of replies the receiver is permitted to make as a response to a particular type of query. Furthermore, we see is it as necessary to develop a standard library of problem-specific protocols (e.g., for task distribution, resource allocation, bargaining, etc.), if heterogeneous agents are to undertake any sophisticated interactions with one another.

Finally, real intelligence in agents may be achieved if agents could autonomously reason about what and with whom to communicate under given circumstances. This requires the agents to be able to reason about communicative actions the way they reason about other actions. A model of such a reasoning mechanism may be realistically incorporated when an agent's

behaviour is described in terms of the inter-relationship of some intentional notions such as the agent's *beliefs, goals, actions, intentions, plans*, and *preferences*. A well-known model for practical reasoning, that is reasoning about actions and goals in a dynamic environment, is the *belief, desire, intention* (BDI) architecture [8]. How agents may reason about communication and their interactions with other agents, in such a model, is described in [9], which also gives the conceptual and theoretical foundations of practical reasoning and Coordination protocols.

4 Summary

Mobile agent technology will be a powerful mean to realise innovative network products. The field is still maturing and is still far from a wide spread acceptance. As with many other new technologies, many applications have to be developed and tried until the technology gradually matures and accepted. Until then, there are still many measures to be taken and this paper was an attempt to project some of these requirements and cite some of the current activities in this area.

Furthermore, this paper argued that for a real intelligent applications the mobile agent technology should evolve and look back at the findings and developments in multi-agent systems and in general AI where the *agent* has been long studied from philosophical, psychological, sociological, anthropological and computational points of view to avoid "re-defining the wheel". Among these works, this article made references to agent architectures, in particular the BDI architectures. Furthermore, we argued that for agents to be able to interact efficiently and coordinate their knowledge and activities, they must be able to undertake meaningful communication and set up meaningful dialogues. For this purpose we cited our current activities coordination protocols.

References

[1] T. Oates, M. Nagendra Prasad, V. Lesser, and K. Decker. A Distributed Problem Solving Approach to Cooperative Information Gathering. *AAAI Spring Symposium on Information Gathering from Distributed, Heterogeneous Environmants*, 1995. `http://www.isi.edu/sims/knoblock/sss95/proceedings.html`

[2] B. Burmeister, A. Haddadi, and K. Sundermeyer. Configurable Cooperation Protocols for Multi-Agent Systems. In C. Castelfranchi and J. -P. Müller, editor, *From Reaction to Cognition*. LNAI 957, Springer, 1993.

[3] D. M. Chess, C. G. Harrison, and A. Kerschenbaum. *Mobile Agents: Are they a good idea?*. IBM research report no. RC 19887, 1994.

[4] T. Finin and J. Weber and G. Wiederholt and M. Genesereth and R. Fitzson and J. McGuire and S. Shapiro and C. Beck. *DRAFT Specification of the KQML Agent-Communication Language*, 1993.
`http://www.cs.umbc.edu/kqml/kqmlspec/spec.html`

[5] David Flanagan. *Java in a Nutshell*. O'Reilly & Associates, 1996.

[6] D. K. Gifford, and J. W. Stamos. *Remote evaluation.* ACM Transactions on Programming Languages and Systems, vol2, no. 4, 1990.

[7] J. Gosling and H. McGilton. *The Java Language Environment: A White Paper.* Sun Microsystems Computer Comapany. October 1995.

[8] A. Haddadi and K. Sundermeyer. *Belief, Desire, Intention Agent Architectures.* In N. Jennings and G. O'Hare, editors, *Foundations of Distributed Artificial Intelligence.* Wiley Inter-Science, 1996.

[9] A. Haddadi. *Communication and Cooperation in Agent Systems: A Pragmatic T heory.* Springer, LNAI series, No. 1056, 1996.

[10] D. Kotz, R. Gray, and D. Rus. *Transportable Agents Support Worldwide Applications.* In Proceedings of SIGOPS-96, 1996.

[11] Jon Siegel. *CORBA Fundamentals and Programming.* Wiley, 1996.
`http://www.acl.lanl.gov/CORBA/#BOOKS.`

[12] J. E. White. *A high-level framework for network-based resource sharing.* In Proceedings of AFIPS conference, 1076.

[13] James White. Telescript Technology: Mobile Agents. In Bradshaw and Jeffrey (ed.) *Software Agents.* Menlo Park, California: AAAI Press/The MIT Press, 1996.

[14] *CyberAgent,* Ftp Software Inc., 1996.
`http://www.ftp.com/cyberagents/cyberftp.html`

[15] *Mole,* University of Stuttgart, Germany, 1996.
`http://www.informatik.uni-stuttgart.de`
`/ipvr/vs/projekte/mole.html`

[16] *Java Agent Template,* Stanford University, CA, 1996.
`http://cdr.stanford.edu/ABE/JavaAgent.html`

[17] *Java-To-Go,* Berkeley Univerity, CA, 1996.
`http://ptolemy.eecs.berkeley.edu/~wli/group/java2go`
`/java-to-go.html`

[18] *Agent Builder,* IBM, U.S.A., 1996.
`http://www.raleigh.ibm.com/iag/iagsoft.htm`

[19] *Aglets,* IBM, Japan, 1996.
`http://www.ibm.co.jp/trl/projects/aglets/`

[20] *Kona,* SAIC, U.S.A., 1996.
`http://riz.saic.com/AIT//agentref.html`

[21] *Tabriz,* General Magic, U.S.A., 1996.
`http://www.genmagic.com/Tabriz/`

Coordinating Active Agents in Open Systems

Thilo Kielmann
University of Siegen,
Dept. of Electrical Engineering and Computer Science
Hölderlinstr. 3, D–57068 Siegen, Germany
kielmann@informatik.uni-siegen.de

Abstract

The field of agent oriented programming (AOP) recently emerged from its object–oriented roots. Whereas the benefits of object–orientation nowadays are widely used in AOP, modeling of communication in multi–agent systems (MAS) still lacks adequate abstractions. Currently considered approaches like RM–ODP, CORBA, or languages like Java introduce object–orientation to inter–agent communication while unfortunately providing only rather low–level communication abstractions.

This work introduces Objective Linda, a coordination model especially designed for the needs of communication between active agents in open systems. We present how Objective Linda can be used as a suitable platform for MAS and illustrate this on an example of a traffic scenario.

1 Introduction

The notion of *Agent Oriented Programming* (AOP) [Sho93] has been coined as a specializing evolvement of actor systems which today manifest the de–facto model for systems of communicating active objects [Agh86]. In AOP, objects are specialized to agents while object state is treated as the agent's *mental state* and where communication between agents is typically considered in terms of *speech act theory* [Sea69] like *information, request, offer* etc.

In [Bur95], AOP has been shown to be especially beneficial for use in open systems. In that work, open systems are characterized by the following requirements on agents: *continuous availability* (persisting individual operations), *extensibility* (coping with dynamic configurations), *decentral control*, *asynchrony*, *inconsistent information* (lack of globally consistent states), and *arms–length relationships* (coping with only local and hence incomplete system knowledge).

In this work, we will outline our notion of open systems which comes close to the above–mentioned definitions. We then argue that mere object–orientation is not sufficient for adequately modeling the interoperation of active agents. Instead, so–called *coordination models* should be used to express and constrain object interactions. Then we present our coordination model Objective Linda and illustrate its usefulness for the implementation of multi–agent systems (MAS). Hence, our contribution is a coordination platform on top of which intelligent agents in the sense of AOP can be implemented.

The paper is structured as follows: In Sect. 2, we will clarify our notions of coordination and of open systems and we will briefly evaluate commonly used approaches. In Sect. 3, we present our coordination model Objective Linda. Its usefulness for implementing active agents will be shown by an example in Sect. 4.

2 Communication Platforms for Open Systems

Open systems are systems in which new active entities ("objects", "agents", or "actors") may dynamically join and leave, i.e. evolving, self–organizing systems of interacting agents [Agh86, Cia90]. It is widely accepted that open systems are composed of software components which are *encapsulated* and *reactive* [Weg93]. Components are called encapsulated if they have an interface that hides their implementation from clients; they are called reactive if their lifetime is longer than the processing of their atomic interactions (e.g. messages). This definition of components directly leads to object–based design because objects are by their very nature encapsulated and reactive entities.

A fundamental property of open systems is their ability to cope with incremental adaptability. In this perspective, encapsulation captures spatial incrementality by controlled propagation of local state changes and reactiveness enables temporal evolution by incrementally executing interactions. Another fundamental property of open systems is their inherent heterogeneity. The openness for new components implies openness for so–far unknown kinds of components yielding systems which are composed of various kinds of hard- and software.

Programming open systems is primarily concerned with the *coordination* of concurrently operating active entities. Coordination involves the management of the communication between these entities. Coordination models based on *generative communication* are considered the most prospective approaches to this research domain. Generative communication, as initially introduced in [Gel85], is based on a shared data space, sometimes also called a *blackboard*, in which data items can be stored ("generated") and retrieved later on.

This kind of communication inherently uncouples communicating agents: a potential reader of some data item does not have to take care about it (e.g. as with rendezvous mechanisms) until it actually needs it. The reader does not even have to exist at the time of storing. The latter point leads to the other major advantage of generative communication: agents (the active entities) are able to communicate although they are anonymous to each other. This uncoupled and anonymous communication style directly contributes to the design of coordination models for open systems: uncoupled communication enables to cope with dynamically changing configurations in which agents move or temporarily disappear. Anonymous communication allows to communicate with unknown agents. Hence it allows communication with incomplete knowledge about the system configuration which is a crucial demand of open systems. Due to this fact, coordination models based on generative communication are superior to message passing or trader–based schemes because these both rely on knowledge about a receiver's or server's identification.

A related important notion is the one of *open distributed systems*. It is defined in the upcoming ISO reference model of open distributed processing (RM–ODP) [ISO95]. In the RM–ODP definition, *distributed systems* have to cope with *remoteness* of components, with *concurrency*, the *lack of a global state*, and *asynchrony* of state changes. In addition, *open distributed systems* are characterized by *heterogeneity* in all parts of the involved systems, *autonomy* of various management or control authorities and organizational entities, *evolution* of the system configuration, and *mobility* of programs and data.

The RM–ODP model which conceptually provides the basis for commercially available systems uses object–based modeling too; also because of the principal object properties of encapsulation and reactiveness. RM–ODP focuses on interaction between objects based on the client/server architecture: "They (objects) embody ideas of services offered by an object to

its environments, that is, to other objects." [ISO95] In RM–ODP, coordination between objects takes place via centralized instances, so called *traders* [ISO94], which are repositories of service type definitions, used to identify offered and requested services.

Presumably the most prominent commercial system for open, object–based systems is the Common Object Request Broker Architecture (CORBA) [Obj93]. Its central component, the Object Request Broker (ORB) acts as a trader in the sense of RM–ODP. Like other traders, the ORB provides references to server objects which in case of dynamically changing configurations may quickly turn into void ("dangling") references causing problems in open configurations. Today, client/server architectures are seen as the current intermediate step on the way from mainframe–oriented to collaborative (peer–to–peer) computing [Lew95]. Nevertheless, service–oriented communication is an important paradigm for open distributed systems [Adl95] and must hence be captured by coordination models. But because client/server communication is restricted to the exchange of request/reply pairs, other communication forms like e.g. for group communication can not be modeled adequately. Hence, coordination models for open systems need to be more general in their applicability.

As an alternative approach to object interoperability by trader–based schemes, the programming language *Java* [AG96] recently attracted broad attention. Java is a fully–featured object–oriented programming language with concurrency abstractions based on a thread concept. It's major benefit is a tight coupling to the World Wide Web (WWW) for which a mechanism for dynamic software loading across physically distributed and potentially heterogeneous systems has been developed. This mechanism, together with Java's interpreted code execution, enables the development of mobile code which is a crucial feature for the vision of autonomous software agents roaming around the Internet.

Unfortunately, the communication abstractions provided by the Java runtime system are rather low–level, like datagrams, sockets, and a wrapper to access documents in the Web. There are no suitable abstractions for expressing behaviour and interactions of active agents. This is where generative coordination models come into play, as we already outlined above.

3 Objective Linda

We will now introduce the coordination model Objective Linda which we use as the basis of this work. A complete description can be found in [Kie96a]. Objective Linda is based on the foundations of Linda and has been designed in order to meet the requirements of open systems. We start with its language–independent object model, then outline how multiple object spaces can be handled cleanly in open systems, and complete by presenting the set of operations on object spaces.

3.1 Objective Linda's Object Model

Since the goal is to model open systems, a language–independent object model is necessary. In Objective Linda, objects to be stored in *object spaces* are self–contained entities; their interface operations only affect their encapsulated object state. The objects are instances of abstract data types which are described in a language–independent notation, called *Object Interchange Language* (OIL). Actual programs may hence be written in conventional object–oriented languages to which a binding of the OIL types (e.g. to language–level classes) can be declared. In

OIL, all types form a type hierarchy having a common ancestor called *OIL_object* which defines the basic operations needed by all types. OIL allows subtyping according to the "principle of substitutability" [WZ88] such that an object of type S which is a subtype of T can be used whenever an object of type T is expected.

3.1.1 Object Matching in Objective Linda.

Objective Linda's object model treats objects as encapsulated entities which can only be accessed via their interface routines defined by the corresponding type. Consequently, object matching (the process of identifying objects to be retrieved from object spaces) in Objective Linda is based on object *types* and the *predicates* defined by type interfaces. A potential reader has to specify the type of object it wishes to obtain from an object space and additionally a predicate from the type interface which selects the objects of a given type matching a specific request. Because OIL's subtype relations provide types which can be used as replacements for their supertypes, object matching will also consider objects of subtypes of the requested type.

Denoting the type of objects a reader tries to obtain can be achieved by passing an object as a parameter to the operation. The type of this object can be easily deduced. Passing a predicate is a little bit more difficult. In Objective Linda, the matching predicates are directly integrated into the types on which they operate. Therefore, the type *OIL_object* provides a predicate *match* which takes an object of the same type as parameter and returns a boolean value deciding whether a given object matches certain requirements. Several variants of matching a type can be selected by presetting the encapsulated state of the object provided to a matching operation, which we call a *template object* in the following.

3.1.2 Evaluating Active Objects.

According to Linda's *eval* operation, we will call the activity of an agent the *evaluation* of an active object. In favour of a homogeneous model, passive as well as active objects are characterized by their OIL type. The mechanism used to specify this activity is similar to object matching: the type *OIL_object* provides an operation called *evaluate* whose behaviour is redefined by every type of objects that will become active. Similar to the *match* operation, the behaviour of this operation may depend on the object's state before its evaluation.

In Linda, active tuples are treated as functions and are converted into passive tuples after termination, yielding their results. In contrast with this functional view, Objective Linda treats active objects as encapsulated and reactive agents. Hence, the *eval* operation activates objects which simply disappear after termination. Analogous to Linda, active objects are invisible to operations in charge of retrieving passive objects from object spaces. Hence, the behaviour of agents can only be observed by monitoring the passive objects they produce and consume.

3.2 Multiple Object Spaces in Objective Linda

Configurations in Objective Linda consist of two kinds of objects: (active as well as passive) OIL objects, and object spaces. Active objects have, from the moment of their activation on, access to two object spaces: (1) their *context* which is the object space on which the corresponding *eval* operation has been performed, and (2) a newly created object space called *self* which is

directly associated to the object. With this basic mechanism, hierarchies of nested object spaces can be built providing hierarchical abstractions for sub–configurations.

The restriction to exactly the *context* and *self* object spaces is not powerful enough in order to generally express coordination problems. Therefore, we need a mechanism allowing agents to attach to other, already existing object spaces. This mechanism should reflect that object spaces are not part of agents but are accessed by references. This is necessary because object spaces must by their very nature be shared between agents.

In order to avoid problems with direct (low level) references as well as with global naming schemes, it is necessary to introduce a construct (based on the generative communication mechanism) which allows agents to attach to existing object spaces. Objective Linda therefore introduces a special subtype of *OIL_object* which is called *object space logical. Logicals* combine a reference to an object space with a logical identification such that an object space can be found by matching properties of *logical* objects. These properties can of course be customized to application needs by subtyping.

Agents willing to let others attach to object spaces they are already attached to simply create a *logical* object including the reference to the object space to be made available which also contains a convenient logical identification for that object space. This *logical* is then *out*'ed to an object space. An agent a willing to attach to object space n must call a special operation called *attach* on the object space o in which the corresponding *logical* object for n is stored. This operation has two effects: (1) o verifies that n can be attached to (is reachable, allows attachment, etc.), and (2) returns a reference to n which is locally useful to a.

3.3 Operations on Object Spaces

Besides the adaptation of the Linda model to object-orientation, Objective Linda also provides an improved set of operations on object spaces. Improvements concern on one hand the blocking semantics of operations which can be customized by a timeout parameter. On the other hand, operations take multisets of objects instead of single tuples as in Linda.

3.3.1 Operation Blocking.

The operations in the original Linda model have been designed without consideration of openness. As a consequence, the blocking operations for putting an object into an object space (*out*), for consuming an object (*in*), and reading an object (*rd*) assume unrestricted access to the data space and may hence block infinitely in case of open systems where access to an object space may fail due to transient problems.

Furthermore, semantics of the non–blocking versions of *in* and *rd* (*inp* and *rdp*) imply access to a data space as a whole: these operations are defined to immediately return, indicating a failure when there is no object matching a given request. Their semantics must be slightly modified for open systems: operation failure of *inp* and *rdp* should indicate "no such object could be found (in the moment)", reflecting the fact that synchronization based on the absence of a certain object is impossible in open systems.

In order to allow customization of agent behaviour between immediately failing and infinitely blocking, Objective Linda introduces a *timeout* parameter to all of its operations that determines how long an operation should block before a failure will be reported. It can vary between zero and a value indicating an infinite delay.

3.3.2 Multisets of Objects.

Linda's ability to retrieve only one object at a time from an object space is simple and elegant, but unfortunately too restrictive. It is e.g. impossible to non–destructively iterate over all objects of a certain kind [BWA94]. Additionally, synchronization problems can be dealt with more adequately when multiple objects may be consumed atomically from object spaces. These observations lead to the introduction of multisets of objects as parameters and results of operations on object spaces. *in* and *rd* specify multisets of objects to be retrieved by two parameters, namely *min* and *max*. *min* gives the minimal number of objects to be found in order to successfully complete the operation whereas *max* denotes an upper bound allowing to retrieve (small) portions of all objects of a kind. An infinite value for *max* allows to retrieve all currently available objects of a kind.

While multisets of objects are necessary for *in* and *rd*, they have no substantial benefits for *out* or *eval*. But for consistency and simplicity reasons, we use multisets of objects for all operations on object spaces.

3.3.3 Operation Specification.

We can now specify Objective Linda's operations on object spaces. We use a binding to the C++ language as notation.

bool out (MULTISET *m , double timeout)

Tries to move the objects contained in m into the object space. Returns *true* if the operation completed successfully; returns *false* if the operation could not be completed within *timeout* seconds.

MULTISET *in (OIL_OBJECT *o, int min, int max, double timeout)

Tries to remove multiple objects $o'_1 \ldots o'_n$ matching the template object o from the object space and returns a multiset containing them if at least *min* matching objects could be found within *timeout* seconds. In this case, the multiset contains at most *max* objects, even if the object space contained more. If *min* matching objects could not be found within *timeout* seconds, the result has a *NULL* value.

MULTISET *rd (OIL_OBJECT *o, int min, int max, double timeout)

Tries to return clones of multiple objects $o'_1 \ldots o'_n$ matching the template object o and returns a multiset containing them if at least *min* matching objects could be found within *timeout* seconds. In this case, the multiset contains at most *max* objects, even if the object space contained more. If *min* matching objects could not be found within *timeout* seconds, the result has a *NULL* value.

bool eval (MULTISET *m, double timeout)

Tries to move the objects contained in m into the object space and starts their activities. Returns *true* if the operation could be completed successfully; returns *false* if the operation could not be completed within *timeout* seconds.

OBJECT_SPACE *attach (OS_LOGICAL o, double timeout)

Tries to get attached to an object space for which an *OS_LOGICAL* matching o can be found in the current object space. Returns a valid reference to the newly attached object

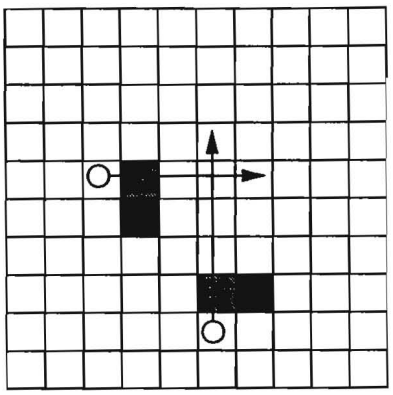

Figure 1: Two cars with intersecting paths.

space if a matching object space logical could be found within *timeout* seconds; otherwise the result has a *NULL* value.

int infinite_matches
Returns a constant value which will be interpreted as infinite number of matching objects when provided as *min* or *max* parameter to *in* and *rd*.

double infinite_time
Returns a constant value which will be interpreted as infinite delay when provided as *timeout* parameter to *out*, *in*, *rd*, and *eval*.

4 An Example: Collision Avoidance

We will now illustrate the suitability of Objective Linda as a platform for implementing multi–agent systems by an example. The scenario described below models the problem of collision avoidance in the traffic domain and has been inspired by the work in [vM92]. Our example is of course overly simplified because our intention is to present Objective Linda as a platform for MAS, rather than intelligent behaviour of the agents themselves.

In our example, agents are concerned with steering cars. Cars drive in a (cyclic) grid which is shown in Fig. 1. Because cars may drive in the four directions *up, down, left,* and *right*, the driveways occasionally intersect. It is the agents' task to avoid collisions in such cases.

We propose a solution in which the agents communicate via an object space. Every agent puts an object of type *Position* into the object space which carries the agent's identification as well as its position on the grid. When changing its position, an agent consumes (using *in*) the *Position* object with its own identification and replaces it by a new one with the updated position.

Whenever an agent wants to make the next step in its desired direction, it has to check first whether it can do so safely. For this purpose, our agents follow the "right–goes–first" priority rule as it is well–known from street traffic. For this purpose, an agent first checks whether it can *rd* a *Position* object for the grid position directly in front of it, in Fig. 1 shown in light–grey colour. If there is such an object, the agent's car will not move but wait. If there is no

```
class Position : public OIL_OBJECT{
private:    bool match_position;    // switching the matching mode
public:     int x,y;                // the grid position
            int car;                // the car's id
bool match(OIL_OBJECT* obj){
    if (match_position)
        return ( (((Position*)obj)->x == x) &&
                 (((Position*)obj)->y == y)    );
    else
        return ((Position*)obj)->car == car;
};
set_match_position() { match_position = true; }
set_match_car()      { match_position = false; }
};
```

Figure 2: Source code of a C++ class *Position*

such object, there is still the possibility of a collision in case a second car will approach on an intersecting path, as it is shown in Fig. 1. For this case, the agent checks for a *Position* object for the grid position in its right–front, in the figure shown in dark–grey colour. Again, if there is such an object, the car stops. Otherwise it moves on. This behaviour is shown as a C++ class *Car* in Fig. 3.

We claim this solution to be adequate for modeling active agents in open systems, because there is no centralized control instance and because agents operate autonomously and asynchronously, and without global knowledge about the number or kinds of cars running in the system. Consequently, agents may join or leave the system at any point of time completely on their own behalf.

Figure 2 sketches the source code of a C++ class *Position* and demonstrates how Objective Linda provides communication abstractions on an adequate level. The focal point of an Objective Linda type for passive objects is its *match* routine. *Position* objects are matched in two different ways: By car id (for updating) and by grid position (for collision avoidance). One can easily see how this can be performed: An agent creates a template *Position* object and sets (as desired) corresponding values either for the car id, or for the position. Finally, it either calls *set_match_position* or *set_match_car* in order to preselect the matching mechanism. Then, the template object can be used as a parameter for *in* or *rd* operations.

Whereas the scenario outlined so far shows the simplest of the possible cases, one can think of several extensions that can be easily supported by Objective Linda:

- A first improvement might be to enlarge the agents' *range of vision*. In order to control an area instead of single grid points, one might easily extend *Position*'s *match* routine to match positions in given intervals. Hence, an agent might retrieve information on all other cars in a certain area within one multiset returned by the *rd* operation.

- One might also consider scenarios with multiple (grid) areas, each represented by a separate object space. These areas might be connected by gates, represented by object–space logicals. Hence, car agents might dynamically change their *context* object space by using Objective Linda's *attach* operation.

```
class Car : public OIL_OBJECT{
private:    int x,y;                  // the grid position
            int car;                  // the car's id
            direction dir;            // the direction to move
void wait(){};   // wait for an arbitrary (random) interval
void evaluate(){
     MULTISET m = new MULTISET;
     Position *p; int nx,ny,px,py;
     while (true) {
         m->put(new Position(id,x,y));
         (void)context->out(m,context->infinite_time);
         wait();
         // store next position to move to in nx and ny
         nx = ... ; ny = ... ;
         p = new Position(id,nx,ny); p->set_match_position();
         m = context->rd(p,1,1,0);
         if ( m ) { // there is a car in front of us
              delete m; delete p;
         }
         else {
              delete p;
              // store position with priority in px and py
              px = ... ; py = ... ;
              p = new Position(id,px,py); p->set_match_position();
              m = context->rd(p,1,1,0);
              if ( m ) { // there is a car with priority
                delete m; delete p;
              }
              else {     // move!
                x = nx; y = ny;
                delete p;
              }
         }
         p = new Position; p->car = id; p->set_match_car();
         m = context->in(p,1,1,context->infinite->time);
         p = m->get(); delete p;
     }
};
};
```

Figure 3: Source code of a C++ class *Car*

- Finally, one might think of systems with different kinds of vehicles that could be represented by objects of different subtypes of *Position*. For purposes of collision avoidance, car agents would still try to *rd* objects of type *Position*. Because Objective Linda's matching considers subtyping, agents could get objects of several subtypes of *Position* in one multiset for a given range.

  For different purposes, agents might look directly for a subtype e.g. in order to answer the question "is there a truck available?"

5 Conclusion

Multi–agent systems need more than the simple (low–level) communication abstractions as they are provided by RM–ODP, CORBA, or languages like Java. Coordination models, esp. Objective Linda which has been designed to meet the requirements of active agents on open systems, provide such abstractions on a higher and hence better–suited level. With the example of collision avoidance given in the previous section, we have illustrated the benefits of our approach.

We are currently experimenting with a prototype implementation of Objective Linda for the C++ programming language based on PVM [GBD+94] as communication platform. First results are encouraging and we have also shown that interoperability between heterogeneous platforms is generally feasible for languages like C++ when communication is based on Objective Linda [Kie96b]. In order to improve interoperability between heterogeneous platforms, we plan to provide the Objective Linda model to Java programs as our next step.

References

[Adl95] Richard M. Adler. Distributed Coordination Models for Client/Server Computing. *IEEE Computer*, 28(4):14–22, 1995.

[AG96] K. Arnold and J. Gosling. *The Java Programming Language*. Addison Wesley, 1996.

[Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. M. I. T. Press, Cambridge, Massachusetts, 1986.

[Bur95] Hans-Dieter Burkhard. Agent–Oriented Programming for Open Systems. In M. J. Wooldridge and N. R. Jennings, editors, *Intelligent Agents, ECAI–94 Workshop on Agent Theories, Architectures, and Languages*, number 890 in Lecture Notes in Artificial Intelligence, pages 291–306, Amsterdam, The Netherlands, 1995. Springer.

[BWA94] Paul Butcher, Alan Wood, and Martin Atkins. Global Synchronisation in Linda. *Concurrency: Practice and Experience*, 6(6):505–516, 1994.

[Cia90] Paolo Ciancarini. Coordination Languages for Open System Design. In *Proc. of IEEE Intern. Conference on Computer Languages*, New Orleans, 1990.

[GBD+94] G. A. Geist, A. L. Beguelin, J. J. Dongarra, W. Jiang, R. J. Manchek, and V. S. Sunderam. *PVM: Parallel Virtual Machine – A Users Guide and Tutorial for Network Parallel Computing*. MIT Press, 1994.

[Gel85] David Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[ISO94] ISO/IEC JTC1/SC21/WG7. Information Technology – Open Distributed Processing – ODP Trading Function. Draft ISO/IEC Standard 13235, Draft ITU–T Recommendation X.9tr, July 1994.

[ISO95] ISO/IEC JTC1/SC21/WG7. Reference Model of Open Distributed Processing. Draft International Standard ISO/IEC 10746–1 to 10746–4, Draft ITU–T Recommendation X.901 to X.904, May 1995.

[Kie96a] Thilo Kielmann. Designing a Coordination Model for Open Systems. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models*, number 1061 in Lecture Notes in Computer Science, pages 267 – 284, Cesena, Italy, 1996. Springer. Proc. COORDINATION' 96.

[Kie96b] Thilo Kielmann. Programming Heterogeneous Workstation Clusters based on Coordination. In *Proc. ICCI'96, 8th International Conference of Computing and Information*, Waterloo, Ontario, Canada, June 1996. Published as special issue of the CD-ROM Journal of Computing and Information (JCI).

[Lew95] Ted G. Lewis. Where is Client/Server Software Headed? *IEEE Computer*, 28(4):49–55, 1995.

[Obj93] Object Management Group. The Common Object Request Broker: Architecture and Specification. OMG Document Number 93.12.43, 1993.

[Sea69] J. R. Searle. *Speech Acts: An Essay in the Philosophy of Language*. Cambridge University Press, Cambridge, England, 1969.

[Sho93] Yoav Shoham. Agent–oriented programming. *Artificial Intelligence*, 60:51–92, 1993.

[vM92] F. von Martial. *Coordinating Plans of Autonomous Agents*. Number 610 in Lecture Notes in Artificial Intelligence. Springer, 1992.

[Weg93] Peter Wegner. Tradeoffs between Reasoning and Modeling. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object–Oriented Programming*, pages 22–41. MIT Press, Cambridge, Mass., 1993.

[WZ88] Peter Wegner and Stanley B. Zdonik. Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like. In S. Gjessing and K. Nygaard, editors, *Proc. ECOOP'88*, number 322 in Lecture Notes in Computer Science, pages 55–77, Oslo, Norway, 1988. Springer.

Java and Agent Oriented Programming

Ralf Kühnel
Institut für Informatik, Humboldt-Universität zu Berlin, Unter den Linden 6, 10099 Berlin
kuehnel@informatik.hu-berlin.de

Abstract

In this paper I will discuss the suitability of the new programming language Java for Agent Oriented Programming. If one roughly defines agents as a combination of distribution, communication, pro-activeness, knowledge representation and problem solving, then Java fits well for the first three points but lacks means for the last two.

1 Introduction

Java is a new programming language ([5]), developed at Sun Microsystems, currently in wide use for applications in the World Wide Web, obviously an open distributed system.

Agent Oriented Programming is a programming technique based on agents as conceptual entities. Agent Oriented Programming can be used for solving problems in programming open distributed systems, such as communication, coordination, by means of Artificial Intelligence, such as knowledge representation, inference and reasoning mechanisms; for connecting expert systems in distributed problem solving scenarios, that means introducing a concept of distribution into AI techniques.

In this paper I will discuss, where Java is appropriate for Agent Oriented Programming and what is missing.

2 Agent Oriented Programming

Agent Oriented Programming as discussed in [8, 7, 2] relies on the assumption, that a complex distributed software system can be programmed as a set of communicating, interacting, knowledge based software entities, called (software) agents. An agent consists of components for perceiving its environment (sensors); for keeping an internal state (e.g. knowledge base); for communicating and interacting with other agents (need for a common language); for establishing and performing actions, based on the current internal state and the environment to achive some goal (e.g. planning, reactive behavior), or by taking the inititiative.

Looking at these four basic properties one can characterize different agent architectures. First the agents internal state can be based on mentalistic notions, such as knowledge, belief, intention. The agent then uses these notions to generate the next action to perform. This is not reflexive but deliberative behavior. The agents (planning) inference procedure computes a set of actions, appropriate to achive some goal, the preferred of which the agent actually performs ([3, 6]).

Alternatively the internal state can be described as a set of (environmental) condition to action rules, or a fixed coupling of sensors to actors (e.g. via artificial neural nets). Selecting an action then means to match the current environment against the rule conditions, or is even not necessary ([1]).

Independent from the specific architecture in most systems the different agent components run in parallel or are intended to run in parallel. So the agent can perform actions while perceiving the environment or receiving messages from other agents.

¿From all the above points one can derive requirements to a programming language, suitable for Agent Oriented Programming. This language has to provide means to express parallelism, to implement communication between processes on different computers, to access sensor data, to represent knowledge, to infer implicite knowledge and to plan actions in order to achieve some goal.

3 Java

If we look at the Java programming language then some of the above requirements are already fullfilled.

But let us first review the language. The most important properties of Java are:

object orientation: Java programms consist of class descriptions, defining the state and the behavior of the class itself and its objects.

independence from operating system, portability: This is achieved by compiling Java source code to Java Byte Code and interpreting it on a Java Virtual Machine, implemented for the most available operating systems.

parallelism: Threads provide a means to express parallel flows of execution using the same address space. Processes are available too.

internetworking: A Java application is able to communicate to another application somewhere on the Internet using the Socket facility. Sockets implement the TCP/IP communication protocoll. Despite of this very simple communication mechanism a Java application is able to communicate to a WWW entity using the http protocoll implemented within the URL class.

The most abstract communication mechanism, currently available, allows for addressing remote objects and remotely invoking their methods. This can be done with the RMI (remote method invocation) package, provided by Sun, or with one of the different CORBA implementations.

database access: The RDBC package provides access to relational databases.

Let us discuss one point in more detail: parallelism. Originally an object oriented programming language provides the concept of passive objects. An Object just reacts on invocation of its methods, no other object will be executed at this time. There is one flow of execution. By implementing threads Java introduces the concept of active objects. A thread is a single flow of control, running in parallel to other threads using the same address space. So we can have more than one active object at the time during runtime.

On single processor machines one needs, of course, a mechanism for simulating parallelism: a scheduler. The Java runtime system contains a scheduler based on thread priorities in contrast to time slicing. This is sometimes annoying because the programmer itself has to take care,

wether all threads get executed some time. In particular he has to programm friendly threads releasing control within an appropriate time by one of four methods to get in a blocked state.

The common address space allows for an easy way of inter-thread-communication: the use of shared data. A synchronization mechanism is, of course, needed in this case. The Java runtime system therefore provides the means of monitors, guaranteering that only one thread at a time is allowed to access some code block.

4 Discussion

Let us match the requirements for Agent Oriented Programming against the properties of Java and discuss some other proposed languages.

Obviously Java fits well the distribution, parallelism and communication requirements. The different agent parts can be implemented as threads. The main thread will be the reasoning and planning thread. The internal state should be accessible from all threads.

In case of communication probably a new layer is required implementing a protocoll for interaction between agents, This layer has to ensure that agents use the same representation language and the same context ontology. If this layer will be built using the Java socket classes, then a naming service is required too. In fact there already exists a collection of classes implementing an agent template based on sockets supporting KQML (Java Agent Template by H.R. Frost, Stanford University). If it instead will be implemented upon a CORBA implementation the naming service is for free.

So Java supports the distribution and parallelism part of agents. But Java does not give us direct support for knowledge representation, reasoning and planning. There are no classes included in the JDK for this purpose. Java is missing the AI part of agents. But there is no reason, why this could not be implemented.

In Shohams Agent-0 ([7]) just this part is most important. He takes a well established AI language such as Lisp and Prolog as a basis and simulates communication, parallelism and distribution. So this language lacks at least this agent part. Of course, it is possible to implement this part using a more suitable language and then import it into the AI language. Obviosly these means lie beyond the original intention of those languages and work within side effects.

Just the other extrem we will get using java, and we can see it already in some systems, for instance the TUB-MAGIC system implemented in Smalltalk. Distribution and parallelism are simulated, however. But it shows another interesting aspect of agents: persistency. The internal agent state can be saved and recovered on restarting the system. This feature of object persistency will be available in future JDK releases too.

So it is possible to improve a language suitable for parallelism, distribution and communication by AI techniques instead of improving an AI language by means of parallelism, distribution and communication. If we take for instance the communication mechanisms required by Agent-0 or Cuncurrent METATEM, then their implementation is very easy in Java.

In order to access sensor data one probably has to implement drivers in a lower level language, like C. Fortunately Java provides a way to include foreign code.

There are some additional aspects of Java, usefull for Agent Oriented Programming. First it is (at least should be) independent from the underlying operating system. The Java network classes provide means to implement service and information agents. Applets provide a standard interface (WWW browser) for accessing agents (personal assistent). Methods for object

migration support the implementation of mobile agents.

5 Conclusion

From the above statements it is clear, what is missing and what have to be done. The above mentioned agent template can only be a first step. At least it implements a communication layer for KQML being more suitable for agent interaction. But this layer should be built upon CORBA or RMI and not upon sockets. What is completely missing in this template and as already said in Java is the agents AI part.

So different representation and reasoning mechanisms should be implemented by means of object orientation. An Agent Oriented Programmer can then choose the most appropriate one for his actual problem. The agents should use a standard agent communication language, like KQML. Such message layer should use techniques from distributed object oriented programming, such as CORBA or ODP, and not reinvent the wheel.

However, AI techniques could improve broker or trader architectures for instance by content based search for finding appropriate services.

References

[1] R.A. Brooks. Intelligence without representation. Artificial Intelligence, 47:139-159, 1991.

[2] H.D. Burkhard. Agent oriented programming for open systems. in Proceedings of ECAI'94 Workshop on Agent Theories, Architectures and Languages. Lecture Notes in AI 890, Springer Verlag, 1995.

[3] P.R. Cohen and H.J. Levesque. Intention is choice with commitment. Artificial Intelligence, 42:213-261, 1990.

[4] M. Fisher. A survey of Concurrent METATEM - the language and its applications. in Temporal Logic - Proceedings of the First International Conference. D.M. Gabbay and H.J. Ohlbach eds. Lecture Notes in AI 827, Springer Verlag, 1994.

[5] R. Kühnel. Die Java-Fibel. Addison-Wesley, 1996.

[6] A.S. Rao and M.P. Georgeff. Modeling rational agents within a BDI-architecture. in Proceedings of Knowledge Representation and Reasoning, 473-484. R. Fikes and E. Sandewall eds. Morgan Kaufman Publishers, 1991.

[7] Y. Shoham. Agent-oriented programming. Artificial Intelligence 60. 1993.

[8] M. Wooldridge and N.R. Jennings. Agent Theories, Architectures, and Languages: A Survey. Proceedings of ECAI'94 Workshop on Agent Theories, Architectures and Languages. in Lecture Notes in AI 890, Springer Verlag, 1995.

ALP: A programming language for reactive intelligent agents

Thomas Weiser
Fakultät für Informatik, TU München, 80290 München
weiser@informatik.tu-muenchen.de

Abstract

ALP is a logic-based language for modelling intelligent behaviour in a dynamic environment. Originated in the tradition of production systems both the recognition and action phases are substantially improved. An incremental bottom-up reasoning mechanism enables the recognition of complex situations in a changing world. Situations are described in a purely declarative manner by means of a Horn clause program. This logic-based component is embedded in a concurrent procedural language, which serves to describe the corresponding reactions of the agent.

1 Introduction

Production systems are widely used as tools to build expert systems, where they act as a decision making system, mostly in a static domain. In the last few years they gained increasing attention in distributed artificial intelligence as a basic cognitive model for intelligent agents [5]. Again the rules are the basic building blocks for the decision making process: how should the agent react in a certain situation.

An example for the use of a production system is the multi agent test-bed Magsy [2]. Each agent is an OPS5 [3] interpreter extended with the capability of asynchronous message passing. Every agent has its own autonomous control and local knowledge and communicates through sending facts to one another. This system has been used for building a distributed planner for flexible manufacturing plants, in which the each machine is modelled as an agent. (Another application of Magsy can be found in [7].)

This is an example that shows the suitability of production systems for modelling reactive behaviour. An agent is part of a dynamic environment. It continuously analysis its situation, activates its own goals and acts according to them. Since the further development of the environment is unpredictable in principle, the agent must be prepared for a variety of possible events and has to be able to react with adequate behaviour patterns. The recognize-act cycle of production systems make them well suited for event-driven programming, which is an important basis for building reactive agents.

But production systems suffer from two substantial drawbacks, which restrict their usefulness for the mentioned applications:

1. The rule selection process utilizes a very simple pattern matching concept with little expressive power. The condition parts of the rules are composed solely of fact patterns as primitives. There is no concept to abstract condition expressions under a new name. So one cannot compose complex expressions out of other expressions. Consequently one cannot use recursive formulas to select a rule.

2. The action parts of the rules are simple sequences of actions without any control structures. Complex procedural operations have to be scattered to several rules, whereby the user is forced to manage the execution context by her own.

In this view OPS5 is a completely unstructured language, regarding both the description of conditions and the formulation of procedural actions. Therefore any larger program gets very hard to manage, since it consists of one large flat rule set with no obvious inner structure. (The early visions for production systems, that each rule is an independent source of knowledge, that their interplay emerges without additional effort and that complex problems can be solved without describing procedures, soon turned out to be not very realistic.)

The simple condition language has another disadvantage. The situations to be recognized by the agent are in general too complex to be expressed in the condition part of a single rule. Thus there is need for firing rules just to do the situation recognition. As a result complex situations cannot be described declaratively. Since OPS5 has no built-in construct to undo the effects of a rule firing, the user has to provide additional rules to monitor and maintain the recognized situations. (Think about maintaining the transitive closure of a changing relation.)

With ALP (*Agent Logic Programming*) we propose a new architecture. It preserves the advantages of production systems (reactive, symbolic, event-driven computation) and introduces new concepts to overcome the drawbacks mentioned.

An ALP process consists of two conceptual components (see figure 1). The first part handles knowledge abstraction and situation recognition. They are described by means of a Horn clause logic program. This program is evaluated by a bottom-up inference engine according to a purely declarative semantics. This logic-oriented component of ALP (which we refer to as the ALP *knowledge base*) infers continuously the set of deducible facts from a varying set of asserted facts.

The second part is the procedural control component. It executes a concurrent imperative program and describes the actions to take in the individual situations. These actions are triggered by the recognition of corresponding situations and in turn modify the facts in the knowledge base.

To be linked with the outside world the agent needs capabilities to perceive and to act. Perceived information is stored as messages in the knowledge base. External actions are effected through special primitives in the procedural part.

2 ALP Knowledge Base

The ALP knowledge base applies Horn clause logic with negation as failure and function symbols in order to handle knowledge representation and abstraction, situation recognition and decision making. It consists of a logic program, a fact base and a forward-chaining inference machine.

The basic expressions of the logic language are predicates, which come in three flavors: Extensional predicates are containers for those facts that may be asserted or retracted through actions or perception. Intensional (or derived) predicates are defined by the clauses of the logic program and are interpreted by the deduced facts. Built-in predicates provide for some basic functions, e.g. arithmetic operations. Accordingly the fact base contains two sets of facts, asserted and deduced ones.

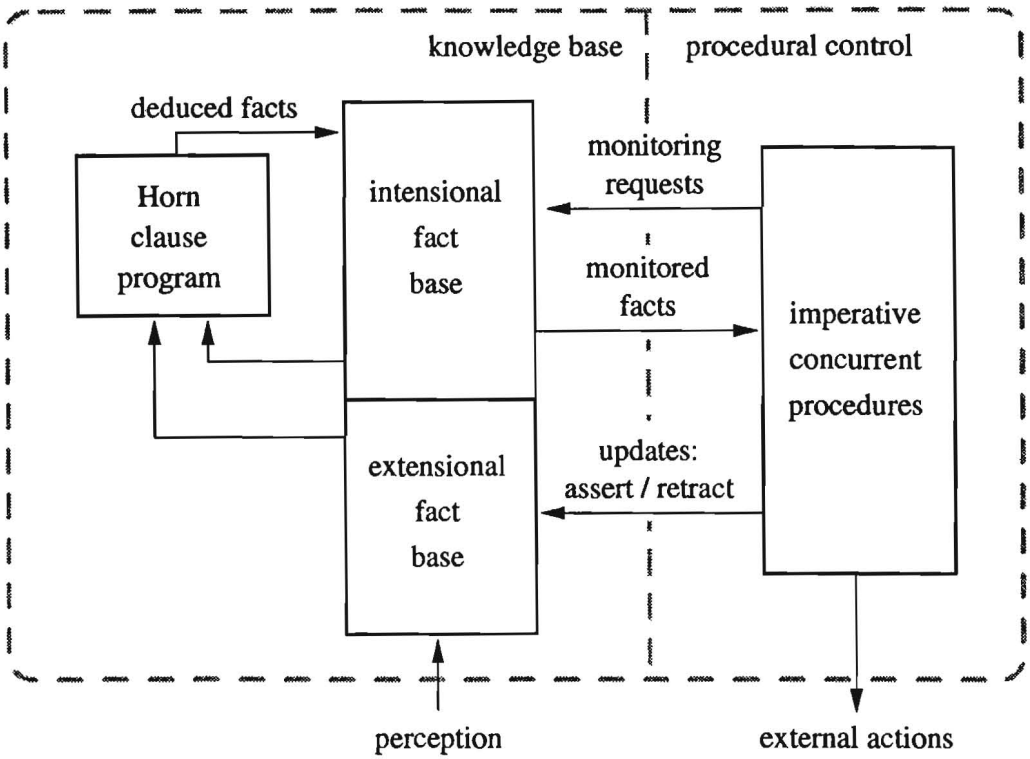

Figure 1: Architecture of an ALP process

The inference machine continuously maintain the set of deduced facts in dependence of the current set of asserted facts and in correspondence to the logic program. This is an incremental and active reasoning process. All changes in the extensional part of the fact base will cause corresponding changes in the intensional predicates. This active bottom-up processing is an essential property for obtaining reactive, event-driven agent behaviour. In contrary to production rules, the clauses (or rules) of the knowledge base have a logical meaning. They have conclusions instead of actions and the conclusions are only valid as long as their premises are.

To bring things in relation to OPS5, the logic program corresponds to the set of all condition parts of the production rules, the extensional fact base corresponds to the working memory and the intensional fact base can be compared with the conflict set of OPS5.

The main difference to OPS5 is that derived predicates now have names and can be used in the definition of other predicates. This has two effects: Firstly, predicates can be written in a more structured fashion in the sense, that you can express complex situations in terms of simpler situations instead of being forced to express everything in terms of extensional predicates. Secondly and even more important, this opens the ability to define recursive predicates, which greatly improves the expressive power.

In the following example, it is assumed that `human` and `par` are extensional predicates. The two clauses define the same-generation relation based on the parent relation.

```
sgc(X,X) ← human(X).
sgc(X,Y) ← par(X,X1), sgc(X1,Y1), par(Y,Y1).
```

The knowledge base supports two types of queries: snapshot queries return the actual fact set of a predicate; monitor queries are requests to inform the client about every change of the monitored predicate. The latter type enables reactive behaviour in the corresponding situations, as the emergence of a fact of a monitored predicate may trigger suitable actions to perform in the recognized situation.

3 The Inference Process

Bottom-up evaluation is a current research topic in deductive databases [8]. One difference is that the ALP knowledge base operates in main memory instead of secondary storage. Moreover, the ALP inference process employs an active incremental algorithm whereas deductive databases usually process queries on request, one after another and without saving intermediate results. In spite of those differences we can make use of some results of the research in deductive databases: we adopt the *well-founded semantics* and we employ *magic set* transformations to speed up the evaluation.

To define the meaning of a Horn clause logic program, several model-theoretic semantics have been studied. The minimal Herbrand model is the most basic one. It applies only to programs without negation. More general, if a program uses negation only outside of recursive paths, the program is called stratifiable. In this case the perfect model semantics supplies the program with a natural meaning.

These restrictions are overcome by the well-founded models semantics [4]. It allows arbitrary combinations of negation and recursion. In this sense it is the most universal one, though this generality has its price. Some programs only have a partial model, meaning that some facts may have an undefined truth value (e.g. in the program { `p(a)` ← ¬ `p(a)` . } the fact `p(a)` is regarded neither true nor false). We believe that this is no real restriction in practice, so we choose this semantics for the ALP knowledge base evaluation process.

The evaluation process is realized basically as an extension of the well-known RETE algorithm [3]. In a first step the logic program is translated into an equation system of relational algebra. Then this system is mapped onto a directed graph, where the nodes are either algebraic operations or places to store the corresponding relations. The graph can be seen as a directed constraint network. As soon as one relation is modified, these changes are propagated through the network until it is stable again, meaning that all equations are satisfied. This method realizes the required activeness and incrementallity of the deduction process.

In the presence of recursive defined relations the algorithm has to be extended in two ways. In the case of recursion without negation a mechanism has to ensure, that there do not remain facts supported solely by themselves without a valid derivation (this is a typical reason maintenance problem). Whereas recursion with negation needs to be handled according to the definition of the well-founded semantics. We have developed a extended version of the RETE algorithm that handles both cases. As this goes beyond the scope of this paper, we omit the details here.

Another technique we adopt from deductive databases is the magic set transformation [1] – with the following background. Compared to top-down evaluation the bottom-up approach has one basic drawback mainly effecting its efficiency: it is not goal directed. A naive bottom-up evaluation generates the complete model of the logic program with respect to a given extensional fact base, regardless whether the generated facts are relevant for the current queries or not.

Nevertheless, bottom-up evaluation can be extended to behave in a goal directed manner. The key idea is to distinguish between input and output arguments of a predicate. The intention is, to generate only that part of the corresponding relation that matches a given set of input values. These input values may be known from the query; or they may be obtained while evaluating the body of a clause: after generating the answer sets for some subgoals this information is passed sideways to constrain the input arguments of the remaining subgoals. This reduce the number of generated facts dramatically without effecting the query result. Moreover, this enables us to handle infinite relations, as long as they are finite for given input values. Lastly, predicates can now be thought of and used as procedures or functions that map their input values to a set of output values.

Evaluation of a predicate should take advantage of bound input arguments. For this the constraints on the input arguments have to be pushed backward through the clauses as far as possible in order to inhibit the generation of unnecessary facts. This idea can be realized by program transformations during compile time. They are known as the family of magic set transformations. There has been much research effort to develop transformations which work even in the presence of recursion and negation [6]. We believe that the application of these techniques will have a great impact on the efficiency and usability of our system.

4 Procedural Control

So far the ALP knowledge base serves as a powerful tool to recognize complex situations. It has to be complemented with procedural concepts to describe the actions to take in those situations.

One possibility is to use the knowledge base as a library in a conventional imperative programming language, e.g. C++. In general such languages are not very well suited for symbolic, event-driven programming.

Alternatively one can follow the traditionally production system approach by linking sequential action scripts to some of the intensional predicates. As mentioned before, this architecture lacks the concept of an execution context. So the user must manage this context by her own to link the pieces of a complex procedural structure together.

According to this we propose to introduce a special procedural language into ALP. Until now we have not defined this language in detail, so we list only some of the intended features here.

The primitive actions available are modifications of the knowledge base, i.e. assertions and retractions of extensional facts. Furthermore, external actions like sending a message or effecting the physical world have to be included.

Control structures on the other hand make the further progress of the procedure dependent on the result of knowledge base queries. In addition, the results of the queries can be bound to variables, which in turn can be used in subsequent actions or control structures.

Control structures can have an immediate or a waiting semantics, depending on whether they employ a snapshot query or a monitor query. The former correspond to constructs in sequential programming languages, like *if–then–else*, the latter are closer to the rule concept of production systems.

Additionally, the language should provide for concurrency constructs, like thread generation, termination and prioritized scheduling.

Procedures containing these constructs can be translated into equivalent sets of production

rules. So these language constructs can be treated as abbreviations or macros that automatically manage execution context and thread scheduling.

5 Conclusion

The ALP architecture combines a deductive knowledge base with a concurrent procedural control component. This structure reflects a basic model for intelligent agents. On the one side an agent has to represent its current beliefs about the world and itself. This knowledge has to be represented on different abstraction levels. Higher levels model the agent's view of its situation and current goals. The ALP knowledge base is a tool to describe such abstraction processes by Horn clause logic in a purely declarative manner. On the other side an agent has to change the world as well as its own beliefs and intentions. Procedures are a natural way to describe these active aspects. We think that the presented combination of declarative and procedural concepts results in a well suited programming model for reactive, intelligent agents.

References

[1] C. Beeri and R. Ramakrishnan. On the Power of Magic. In *Proceedings of the Sixth ACM PODS Symposium on Principles of Database Systems*. 1987.

[2] K. Fischer. The Rule-based Multi-Agent System MAGSY. In *Proceedings of the CKBS'92 Workshop*. DAKE Centre, Keele University, 1993.

[3] C. L. Forgy. RETE: A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem. *Artificial Intelligence* 19. 1982.

[4] A. Van Gelder, K. A. Ross and J. S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM* 38(3). 1991.

[5] T. Ishida. Parallel, Distributed and Multi-Agent Production Systems. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*. San Francisco, CA, 1995.

[6] D. B. Kemp, D. Srivastava and P. J. Stuckey. Bottom-Up Evaluation and Query Optimization of Well-Founded Models. *Theoretical Computer Science* 146(1&2). 1995.

[7] J. P. Müller and M. Pischel. The Agent Architecture InteRRaP: Concept and Application. Technical Report RR-93-26, DFKI Saarbrücken, 1993.

[8] R. Ramakrishnan and J. D. Ullman. A survey of deductive database systems. *Journal of Logic Programming* 23(2). 1995.

IPDL - Interaction Protocols for Distributed Objects

Boris Bokowski
FU Berlin, Institut für Informatik, Takustraße 9, D-14195 Berlin
bokowski@inf.fu-berlin.de

Abstract

In object-oriented frameworks, the interactions between objects are often complex and difficult to understand. In this paper, "interaction protocols" are introduced as a means to describe object interactions. The interface definition language "IPDL", an extension of OMG's IDL, is introduced for specifying object interactions. Besides making object-oriented system descriptions more readable by explicitly representing inter-object behaviour, interaction protocols can be used to detect programming errors: it is decidable at compile-time whether an object interacts correctly according to an interaction protocol. Moreover, interaction protocols apply to several levels of abstraction, because complex interaction protocols may be composed hierarchically out of simpler ones.

1 Introduction

Modern object-oriented software development focuses more and more on object composition instead of inheritance as the main means to build complex software systems. Frameworks [Joh93] and Design Patterns [G+95] are the two most prominent areas where object composition and the interaction between composed objects play a crucial role. Language support for expressing interactions that are more than one-time operation invocations, though, is lacking.

For example, consider the interface of a *Manager* object for resource management with two public operations *lock* and *unlock*. Typically, interactions between a *Manager* and its clients are such that for each invocation of *lock*, the following invocation at *Manager*'s interface has to be that of *unlock*. Currently, longer-term interactions that span more than one invocation need to be represented using sequences of invocations, with no means to group such sequences at the interface level. Moreover, interfaces of objects take a one-sided perspective: they only describe the operations that are to be performed by a server. There is no support for describing interactions that involve actions of a client, for example a callback of *Manager*, during execution of *lock*, to its client's operation *getID*, in one interface together with *lock* and *unlock*. Rather, a separate interface for appropriate clients has to be introduced, with no apparent connection between *lock* and *unlock* on one hand and *getID* on the other hand. Consequently, other forms of interaction than client/server, as e.g. producer/consumer, model/view/controller, or exporter/trader/importer, are not expressible either.

In this paper, an extension to the CORBA-IDL [OM95], called "IPDL" for Interaction Protocol Definition Language, is proposed for describing generalized interfaces for long-term, high-level, composite, and multi-role interactions.

1. An interaction protocol describes the interaction interface between two or more objects from an external perspective, so that interfaces are independent from single classes, and all involved roles in an interaction are described explicitly.

2. For each object that participates in an interaction, the interaction protocol defines the object's allowed and expected behaviour in terms of causal connections between interactions.
3. Complex interaction protocols can be built by specifying allowed sequences of simpler interaction protocols, which caters for the description of interactions at different levels of abstraction.

2 Interfaces and IDL

In this section, we give a short overview about the CORBA IDL and how it is used in the context of distributed systems.

The interface description language CORBA-IDL is used to describe client/server interfaces. For each interface, a number of operations and attributes are specified which have to be supported by servers implementing that interface. For example, an interface of a *Manager* object that manages the locking of a shared resource would be specified as follows in IDL (see Fig. 1)

```
interface Manager {
    void lock(in MClient c);
    void unlock();
};
```

Fig. 1: *Manager* interface described in IDL

Note that in the IDL itself (i.e., without using comments) there is no way to specify that each invocation of *lock* must be followed by an invocation of *unlock*.

If, for detecting deadlocks, the *Manager* object needed to call a *getID* operation for processing *lock*, the corresponding method signature needs to be included in a separate interface for *MClient* (see Fig. 2), making *getID* accessible for any object that has a reference to a *MClient* object, at any time. It is not possible to express that the ID of a *MClient* may only be requested by a *Manager* during its execution of the *lock* operation. Moreover, the interaction between a *Manager* and a *MClient* object, which was designed by considering both sides at once, has to be described in two separate parts.

```
interface MClient {
    short getID();
};
```

Fig. 2: *MClient* interface described in IDL

The CORBA-IDL describes interfaces of objects that are possibly distributed over several processing nodes. To allow transparent access to distributed objects, IDL files are used for generating stub classes for both the client and the server side. The client-side stub implements a proxy object that from the client object's perspective is indistinguishable from the actual server object. The proxy object forwards all operation requests to a corresponding server-side stub, which implements a driver object that calls the server object, so that from the server object's

perspective, this call is indistinguishable from a local client's call. After the operation is completed by the server object, the operation's return value is forwarded from the driver object to the proxy object, which returns the value to the client object.

For our example of *Manager* and *MClient*, four helper classes would be generated by a CORBA stub generator: one proxy class and one driver class for each of the two interfaces. As we will see in the next section, it is possible to generate only two helper classes, if the allowed interaction between *Manager* and *MClient* was described as one interaction as opposed to two interfaces.

3 Protocols and IPDL

In this section, the Interaction Protocol Definition Language (IPDL), an extension to the CORBA-IDL, is described. To distinguish between a CORBA-IDL interface definition and a IPDL interaction protocol definition, IPDL definitions start with the keyword "interaction" rather than "interface". However, "interface" definitions using the old style are still valid in IPDL, and they implicitly define two roles "Client" and "Server".

There are two main differences between an interface definition and an interaction protocol definition. One is that it is no longer true that for each interface, there is always one client role which issues requests, and one server role which services requests. Rather, multiple *roles* may be defined, and actions such as issuing or servicing a particular request may be assigned to each of those roles. Other actions such as sending or receiving a message, or forwarding a message or a request, may be defined as well. The other difference is that a *protocol* may be given that constrains the order in which actions may happen.

We can convert the example of section 1 into IPDL in two steps: in the first step, the two interfaces are combined into one interaction definition with two roles *Manager* and *MClient*. In the second step, the protocol to be obeyed by the two roles is added.

```
interaction Mutex {
    roles MClient, Manager;
    Oneway(MClient, Manager) void lock();
    Invocation(Manager, MClient) short getID();
    Oneway(Manager, MClient) void lock_granted();
    Invocation(MClient, Manager) void unlock();
};
```

Fig. 3: *Mutex* interaction described in IPDL, version 1

In Fig. 3, the two roles *Manager* and *MClient* are described in one *interaction* definition. The combined interaction is called *Mutex*. The actions (which were "operations" in CORBA-IDL terms) have been annotated by either *Invocation* or *Oneway*, primitive actions for operation invocation and message passing, respectively. For each such action declaration, the participating roles are given in parentheses, for specifying which role is the sender or the receiver of a *Oneway*, and which role is the caller or the callee of an *Invocation*. The reason for splitting the *lock* operation into two parts, *lock* and *lock_granted*, will be explained shortly.

Note that the order in which the action declarations are given does not imply that the actions may only happen in this particular sequence.

To define such a sequence, or more generally, to constrain the possible orderings of actions, a protocol may be defined at the end of an interaction protocol definition. In Fig. 4, the protocol for the proper interaction between *Manager* and *MClient* is given. The language for defining such protocols allows to express them at the level of regular expressions, with constructs for sequences, alternatives, and repetition, in order to make the problem of comparing protocol definitions statically decidable. In this example, sequencing of actions (using ";") and repetition (using "*loop { ... }*") is used. To allow the callback *getID* while the *Manager* processes the *lock* request, the former operation *lock* had to be split into two *Oneway* messages *lock* and *lock_granted.*

```
interaction Mutex {
    ... definition of roles and action declarations ...
    protocol:
    loop {
        lock;
        getID;
        lock_granted;
        unlock;
    };
};
```

Fig. 4: *Mutex* interaction described in IPDL, version 2

To remedy this, IPDL allows parameterization of interaction protocol definitions. In particular, *Invocation*, although being a primitive action, is a parameterized interaction protocol which may be defined as seen in Fig. 5. If no action is given as parameter, the non-action "*nop*" is used as a default for *inner*. It might be surprising that the actions which are used to define an interaction protocol are themselves defined as interaction protocols. This allows interactions to be defined at several levels of abstraction. In fact, even the *Mutex* interaction protocol may be used to build more complex protocols. In the final version of *Mutex* (Fig. 6), to make it more reusable, an "*inner*" parameter is introduced for *Mutex* as well.

```
interaction Invocation<inner=nop> {
    Oneway(Caller, Callee) void arguments(... argument types ...);
    Oneway(Callee, Caller) ... return type ... result();
    protocol:
    arguments;
    inner;
    result;
}
```

Fig. 5: Interaction protocol *Call*

```
interaction Mutex<inner=nop> {
    roles MClient, Manager;
    Invocation(MClient, Manager) void lock();
    Invocation(MClient, Manager) void unlock();
    Invocation(Manager, MClient) short getID();
    protocol:
    loop {
        lock <
            getID;
        >;
        inner;
        unlock;
    };
};
```

Fig. 6: *Mutex* interaction described in IPDL, version 3

Using only sequences, loops, and alternatives may seem too restricted to describe all kinds of protocols. However, more complex protocols may be approximated by our restricted protocol definition language, which was designed to enable static checking of protocol obeyance. To show that non-trivial protocols indeed can be described in IPDL, a second example is given in Fig. 7. *DatabaseAccess* describes a low-level (i.e. non-transparent) interface to an object-oriented database. A *Client*, after registering with a *beginTransaction* message, may issue *read* or *write* requests as long as the Server responds with *accept* and not with *reject* (which indicates that the transaction conflicts with another transaction), and until an *endTransaction* message is sent by the *Client*. In this example, the notation for alternatives is used ("*select { | <branch1> | ... | <branchn> }*"), and the modifier "*[exits]*" is used to denote actions that cause the enclosing *loop* to be ended.

Because all partners of an interaction are known, in contrast to the CORBA-IDL, only one stub class for each role in that interaction has to be generated. If it is required by the protocol, each stub class for a specific role may act both as a proxy and a driver for the object which plays that role. For instance, for *DatabaseAccess*, two stub classes *DatabaseAccessClient* and *DatabaseAccessServer*, shown in Fig. 8, are generated (currently, Java [AG96] is used as the target language). Note that so far, the stub generator generates *explicit* communication operations; the generator will be extended so that it may generate code for *implicit* communication operations as well (such as only one blocking operation for *read* instead of sending and receiving two messages for each *read*). The class *DatabaseAccessClient* will interact with the object playing the *Server* role, while *DatabaseAccessServer* will be the stub class that interacts with the *Client*. The actual code for the generated operations is not shown. It checks whether the specified protocol is obeyed, and provides for the communication with the corresponding other stub.

As mentioned earlier, obeyance of a protocol may be checked at compile-time. The runtime checks which are included in the generated stub classes are needed only when using an existing object-oriented language for the implementation of objects. In the next section, we will discuss how an object-oriented language like Java may be extended to allow for compile-time checking of protocol obeyance.

```
interaction DatabaseAccess {
   roles Client, Server;
   Oneway (Client, Server) void beginTransaction();
   Oneway (Client, Server) void write(in DBObject o, in DBAttribute a, in DBValue v);
   Oneway (Client, Server) void endTransaction();
   Invocation(Client, Server) DBValue read(in DBObject o, in DBAttribute a);
   Oneway(Server, Client) void accept();
   Oneway(Server, Client) void reject();
   {
      loop {
         beginTransaction;
         loop {
            select {
               | accept;
               | reject[exits];
            }
            select {
               | read;
               | write;
               | endTransaction;
                  select {
                     | accept[exits];
                     | reject[exits];
} } } } }     }
```

Fig. 7: Interaction Protocol Definition for *DatabaseAccess*

```
class DatabaseAccessClient extends DatabaseAccessProtocol {
   void receive_beginTransaction() { ... }
   Message receive_read_write_endTransaction() { ... }
   void send_read_result(DBValue result) { ... }
   void send_accept() { ... }
   void send_reject() { ... }
}

class DatabaseAccessServer extends DatabaseAccessProtocol {
   void send_beginTransaction() { ... }
   Message receive_accept_reject() { ... }
   void send_read_request(DBObject o, DBAttribute a) { ... }
   Message receive_read_result() { ... }
   void send_write_request(DBObject o, DBAttribute a, DBValue v) { ... }
   void send_endTransaction() { ... }
}
```

Fig. 8: generated stub classes for *DatabaseAccess*

4 Implementing Objects

While interaction protocols specify sequential interaction behaviour, objects are allowed to play several roles concurrently (or interleaved), so that they can engage in more than one interaction at the same time. Every possible interaction that an object can perform with other objects has to be associated with a role that the object plays in that interaction. This implies that all actions of an object that are visible to other objects must be described in some interaction protocol that specifies corresponding actions in those other objects.

The set of roles that an object plays constitute its interface. Because the different roles of an object are considered to be of equal importance, there is no special treatment for the "callee" role(s) of an object. Thus, the common view of an object as a provider of a set of services, each service represented by a public method of the object, would restrict the roles that an object can play. To overcome this restriction, in the proposed object implementation language, a class specifies the possible sequences of (inter-)actions that its objects are able to perform. Subsequences of interactions can be given a name, to allow reuse of method-level entities where appropriate.

Conformance of an object to a specific role of an interaction protocol can be defined informally as the fulfillment of the other roles' expectations in that interaction. To fulfill these expectations, an object has to obey the following two rules:

1. Whenever, at a given state of the interaction, the object is active (it is the object's choice what subinteraction to perform and when), only one of the subinteractions allowed to occur from that state may be chosen.
2. Whenever, at a given state of the interaction, the object is passive (another object is active), it must be able to handle any subinteraction allowed to occur from that state.

To allow full computability power for objects, an object's implementation must include parts that cannot be taken into account when checking role conformance, such that - as expected - not all protocol errors (deadlock) can be detected statically. However, the behaviour information encoded in the interaction protocols is a good approximation to behaviour for which conformance still is decidable.

In order to check at compile-time whether an object conforms to a role, the descriptions of object behaviour with respect to that role must be reducible to regular expression-like process specifications. Thus object implementations should build upon constructs similar to the building blocks of IPDL protocols: sequence, active and passive selection, and active and passive repetition constructs. Currently, an extension to the object-oriented language Java that contains these constructs is under development. The two main problems that remain to be solved are how to integrate the "normal" implicit acceptance of operation requests by the methods of an object with a more process-oriented view in which the control flow can be extracted easily, and how to handle inheritance.

Note that the proposed model allows *subtyping*: the ability to assign objects of different types, each of which conforms to a more general type, to a variable having this general type. The proposed language should explicitly support subtyping in that object references always are references to objects in a specific role, and not to the objects themselves, and a behaviour-sensitive subtype relation on roles can be defined.

However, inheritance as a means to generate new classes out of existing ones only by defining the incremental changes that have to be applied to the code of the original class, is not very easily supported. One might think of possibilities to allow to specify incremental changes to classes in the proposed model, but, in general, subclassing does not necessarily imply subtyping. In particular, for concurrent object-oriented programming, the proposed model does not solve the *inheritance anomaly*.

5 Related Work

This work is in the tradition of the work on Contracts [H+90, Hol92], in that it allows for the specification of interaction behaviour of objects, such that each object's expectations can be made explicit. Unlike Contracts, the specification of interaction behaviour is designed such that it can be used for static checks whether a given object composition conforms to the behaviour specifications.

At the same time, the work is in the tradition of Oscar Nierstrasz' work on "Regular Types for Active Objects" [Nie93], in that it describes the interaction behaviour in terms of process specifications that can be checked for equivalence at compile-time. This paper takes a new approach to the interfaces of objects, however, in that not only the servers' roles are specified, such that servers can make assumptions about how they are used by clients; additionally, the clients' roles are taken into account as well, and that in general the collective behaviour of groups of objects can be specified, specifying what interaction behaviour each object in the interaction can expect from the other objects.

Additionally, this work is related to research of Allen and Garlan on Wright [AG94a, AG94b], where a software architecture language is defined that captures the protocol between connected software architecture components. Unlike in Wright, interaction protocols are hierarchical in that protocols can be hierarchically nested, which parallels the use of methods in object-oriented programming.

References

[AG94a] Robert Allen and David Garlan. Formalizing Architectural Connection. In *Proceedings of the International Conference on Software Engineering (ICSE-16).* IEEE Computer Society Press, 1994.

[AG94b] Robert Allen and David Garlan. Formal Connectors. *Technical Report CMU-CS-94-115.* Carnegie Mellon University, 1994.

[AG96] Ken Arnold and James Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

[G+95] Erich Gamma and Richard Helm and Ralph Johnson and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995

[H+90] Richard Helm and Ian M. Holland and Dipayan Gangopadhyay. Contracts - Specifying Behavioural Compositions in Object-Oriented Systems. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'90).* ACM Press, 1990

[Hol92] Ian M. Holland. Specifying Reusable Components Using Contracts. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'92)*. Springer Verlag, 1992

[Joh93] Ralph E. Johnson. How to Design Frameworks. *Tutorial Notes, Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'93)*.

[Nie93] Oscar Nierstrasz. Regular Types for Active Objects. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'93)*. ACM Press, 1993

[OM95] Object Management Group, Inc. *The Common Object Request Broker: Architecture and Specification*. Revision 2.0, July 1995 (available electronically at http://www.omg.org/docs/ptc/96-03-04.ps)

Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

-Bibliothek, Information
und Dokumentation (BID)-
PF 2080
67608 Kaiserslautern
FRG

Telefon (0631) 205-3506
Telefax (0631) 205-3210

e-mail
dfkibib@dfki.uni-kl.de
WWW
http://www.dfki.uni-sb.de/dfkibib

Veröffentlichungen des DFKI

Die folgenden DFKI Veröffentlichungen sowie die aktuelle Liste von allen bisher erschienenen Publikationen können von der oben angegebenen Adresse oder (so sie als per ftp erhaeltlich angemerkt sind) per anonymous ftp von ftp.dfki.uni-kl.de (131.246.241.100) im Verzeichnis pub/Publications bezogen werden. Die Berichte werden, wenn nicht anders gekennzeichnet, kostenlos abgegeben.

DFKI Publications

The following DFKI publications or the list of all published papers so far are obtainable from the above address or (if they are marked as obtainable by ftp) by anonymous ftp from ftp.dfki.uni-kl.de (131.246.241.100) in the directory pub/Publications.
The reports are distributed free of charge except where otherwise noted.

DFKI Research Reports

1996

RR-96-05
Stephan Busemann
Best-First Surface Realization
11 pages

RR-96-03
Günter Neumann
Interleaving
Natural Language Parsing and Generation
Through Uniform Processing
51 pages

RR-96-02
E.André, J. Müller , T.Rist:
PPP-Persona: Ein objektorientierter Multimedia-Präsentationsagent
14 Seiten

1995

RR-95-20
Hans-Ulrich Krieger
Typed Feature Structures, Definite Equivalences, Greatest Model Semantics, and Nonmonotonicity
27 pages

RR-95-19
Abdel Kader Diagne, Walter Kasper, Hans-Ulrich Krieger
Distributed Parsing With HPSG Grammar
20 pages

RR-95-18
Hans-Ulrich Krieger, Ulrich Schäfer
Efficient Parameterizable Type Expansion for Typed Feature Formalisms
19 pages

RR-95-17
Hans-Ulrich Krieger
Classification and Representation of Types in TDL
17 pages

RR-95-16
Martin Müller, Tobias Van Roy
Title not set
0 pages

Note: The author(s) were unable to deliver this document for printing before the end of the year. It will be printed next year.

RR-95-15
Joachim Niehren, Tobias Van Roy
Title not set
0 pages

Note: The author(s) were unable to deliver this document for printing before the end of the year. It will be printed next year.

RR-95-14
Joachim Niehren
Functional Computation as Concurrent Computation
50 pages

RR-95-13
Werner Stephan, Susanne Biundo
Deduction-based Refinement Planning
14 pages

RR-95-12
Walter Hower, Winfried H. Graf
Research in Constraint-Based Layout, Visualization, CAD, and Related Topics: A Bibliographical Survey
33 pages

RR-95-11
Anne Kilger, Wolgang Finkler
Incremental Generation for Real-Time Applications
47 pages

RR-95-10
Gert Smolka
The Oz Programming Model
23 pages

RR-95-09
M. Buchheit, F. M. Donini, W. Nutt, A. Schaerf
A Refined Architecture for Terminological Systems: Terminology = Schema + Views
71 pages

RR-95-08
Michael Mehl, Ralf Scheidhauer, Christian Schulte
An Abstract Machine for Oz
23 pages

RR-95-07
Francesco M. Donini, Maurizio Lenzerini, Daniele Nardi, Werner Nutt
The Complexity of Concept Languages
57 pages

RR-95-06
Bernd Kiefer, Thomas Fettig
FEGRAMED
An interactive Graphics Editor for Feature Structures
37 pages

RR-95-05
Rolf Backofen, James Rogers, K. Vijay-Shanker
A First-Order Axiomatization of the Theory of Finite Trees
35 pages

RR-95-04
M. Buchheit, H.-J. Bürckert, B. Hollunder, A. Laux, W. Nutt,
M. Wójcik
Task Acquisition with a Description Logic Reasoner
17 pages

RR-95-03
Stephan Baumann, Michael Malburg, Hans-Guenther Hein, Rainer Hoch,
Thomas Kieninger, Norbert Kuhn
Document Analysis at DFKI
Part 2: Information Extraction
40 pages

RR-95-02
Majdi Ben Hadj Ali, Frank Fein, Frank Hoenes, Thorsten Jaeger,
Achim Weigel
Document Analysis at DFKI
Part 1: Image Analysis and Text Recognition
69 pages

RR-95-01
Klaus Fischer, Jörg P. Müller, Markus Pischel
Cooperative Transportation Scheduling
an application Domain for DAI
31 pages

1994

RR-94-39
Hans-Ulrich Krieger
Typed Feature Formalisms as a Common Basis for Linguistic Specification.
21 pages

RR-94-38
Hans Uszkoreit, Rolf Backofen, Stephan Busemann, Abdel Kader Diagne,
Elizabeth A. Hinkelman, Walter Kasper, Bernd Kiefer, Hans-Ulrich Krieger,
Klaus Netter, Günter Neumann, Stephan Oepen, Stephen P. Spackman.
DISCO–An HPSG-based NLP System and its Application for Appointment Scheduling.
13 pages

RR-94-37
Hans-Ulrich Krieger, Ulrich Schäfer
TDL - A Type Description Language for HPSG, Part 1: Overview.
54 pages

RR-94-36
Manfred Meyer
Issues in Concurrent Knowledge Engineering. Knowledge Base and Knowledge Share Evolution.
17 pages

RR-94-35
Rolf Backofen
A Complete Axiomatization of a Theory with Feature and Arity Constraints
49 pages

RR-94-34
Stephan Busemann, Stephan Oepen, Elizabeth A. Hinkelman,
Günter Neumann, Hans Uszkoreit
COSMA – Multi-Participant NL Interaction for Appointment Scheduling
80 pages

RR-94-33
Franz Baader, Armin Laux
Terminological Logics with Modal Operators
29 pages

RR-94-31
Otto Kühn, Volker Becker, Georg Lohse, Philipp Neumann
Integrated Knowledge Utilization and Evolution for the Conservation of Corporate Know-How
17 pages

RR-94-23
Gert Smolka
The Definition of Kernel Oz
53 pages

RR-94-20
Christian Schulte, Gert Smolka, Jörg Würtz
Encapsulated Search and Constraint Programming in Oz
21 pages

RR-94-19
Rainer Hoch
Using IR Techniques for Text Classification in Document Analysis
16 pages

RR-94-18
Rolf Backofen, Ralf Treinen
How to Win a Game with Features
18 pages

RR-94-17
Georg Struth
Philosophical Logics—A Survey and a Bibliography
58 pages

RR-94-16
Gert Smolka
A Foundation for Higher-order Concurrent Constraint Programming
26 pages

RR-94-15
Winfried H. Graf, Stefan Neurohr
Using Graphical Style and Visibility Constraints for a Meaningful Layout in Visual Programming Interfaces
20 pages

RR-94-14
Harold Boley, Ulrich Buhrmann, Christof Kremer
Towards a Sharable Knowledge Base on Recyclable Plastics
14 pages

RR-94-13
Jana Koehler
Planning from Second Principles—A Logic-based Approach
49 pages

RR-94-12
Hubert Comon, Ralf Treinen
Ordering Constraints on Trees
34 pages

RR-94-11
Knut Hinkelmann
A Consequence Finding Approach for Feature Recognition in CAPP
18 pages

RR-94-10
Knut Hinkelmann, Helge Hintze
Computing Cost Estimates for Proof Strategies
22 pages

RR-94-08
Otto Kühn, Björn Höfling
Conserving Corporate Knowledge for Crankshaft Design
17 pages

RR-94-07
Harold Boley
Finite Domains and Exclusions as First-Class Citizens
25 pages

RR-94-06
Dietmar Dengler
An Adaptive Deductive Planning System
17 pages

RR-94-05
Franz Schmalhofer, J. Stuart Aitken, Lyle E. Bourne jr.
Beyond the Knowledge Level: Descriptions of Rational Behavior for Sharing and Reuse
81 pages

RR-94-03
Gert Smolka
A Calculus for Higher-Order Concurrent Constraint Programming with Deep Guards
34 pages

RR-94-02
Elisabeth André, Thomas Rist
Von Textgeneratoren zu Intellimedia-Präsentationssystemen
22 Seiten

RR-94-01
Elisabeth André, Thomas Rist
Multimedia Presentations: The Support of Passive and Active Viewing
15 pages

DFKI Technical Memos

1996

TM-96-01
Gerd Kamp, Holger Wache
CTL — a description Logic with expressive concrete domains
19 pages

1995

TM-95-04
Klaus Schmid
Creative Problem Solving
and
Automated Discovery
— An Analysis of Psychological and AI Research —
152 pages

TM-95-03
Andreas Abecker, Harold Boley, Knut Hinkelmann, Holger Wache,
Franz Schmalhofer
An Environment for Exploring and Validating Declarative Knowledge
11 pages

TM-95-02
Michael Sintek
FLIP: Functional-plus-Logic Programming
on an Integrated Platform
106 pages

TM-95-01
Martin Buchheit, Rüdiger Klein, Werner Nutt
Constructive Problem Solving: A Model Construction Approach towards Configuration
34 pages

1994

TM-94-04
Cornelia Fischer
PAntUDE - An Anti-Unification Algorithm for Expressing Refined Generalizations
22 pages

TM-94-03
Victoria Hall
Uncertainty-Valued Horn Clauses
31 pages

TM-94-02
Rainer Bleisinger, Berthold Kröll
Representation of Non-Convex Time Intervals and Propagation of Non-Convex Relations
11 pages

TM-94-01
Rainer Bleisinger, Klaus-Peter Gores
Text Skimming as a Part in Paper Document Understanding
14 pages

DFKI Documents

1996

D-96-05
Martin Schaaf
Ein Framework zur Erstellung verteilter Anwendungen
94 pages

D-96-03
Winfried Tautges
Der DESIGN-ANALYZER - Decision Support im Designprozess
75 Seiten

1995

D-95-12
F. Baader, M. Buchheit, M. A. Jeusfeld, W. Nutt (Eds.)
Working Notes of the KI'95 Workshop:
KRDB-95 - Reasoning about Structured Objects:
Knowledge Representation Meets Databases
61 pages

D-95-11
Stephan Busemann, Iris Merget
Eine Untersuchung kommerzieller Terminverwaltungssoftware im Hinblick auf die Kopplung mit natürlichsprachlichen Systemen
32 Seiten

D-95-10
Volker Ehresmann
Integration ressourcen-orientierter Techniken in das wissensbasierte Konfigurierungssystem TOOCON
108 Seiten

D-95-09
Antonio Krüger
PROXIMA: Ein System zur Generierung graphischer Abstraktionen
120 Seiten

D-95-08
Technical Staff
DFKI Jahresbericht 1994
63 Seiten

Note: This document is no longer available in printed form.

D-95-07
Ottmar Lutzy
Morphic - Plus
Ein morphologisches Analyseprogramm für die deutsche Flexionsmorphologie und Komposita-Analyse
74 pages

D-95-06
Markus Steffens, Ansgar Bernardi
Integriertes Produktmodell für Behälter aus Faserverbundwerkstoffen
48 Seiten

D-95-05
Georg Schneider
Eine Werkbank zur Erzeugung von 3D-Illustrationen
157 Seiten

D-95-04
Victoria Hall
Integration von Sorten als ausgezeichnete taxonomische Prädikate in eine relational-funktionale Sprache
56 Seiten

D-95-03
Christoph Endres, Lars Klein, Markus Meyer
Implementierung und Erweiterung der Sprache $\mathcal{ALCP}$
110 Seiten

D-95-02
Andreas Butz
BETTY
Ein System zur Planung und Generierung informativer Animationssequenzen
95 Seiten

D-95-01
Susanne Biundo, Wolfgang Tank (Hrsg.)
PuK-95, Beiträge zum 9. Workshop „Planen und Konfigurieren", Februar 1995
169 Seiten

Note: This document is available for a nominal charge of 25 DM (or 15 US-$).

1994

D-94-15
Stephan Oepen
German Nominal Syntax in HPSG
— On Syntactic Categories and Syntagmatic Relations —
80 pages

D-94-14
Hans-Ulrich Krieger, Ulrich Schäfer
TDL - A Type Description Language for HPSG, Part 2: User Guide.
72 pages

D-94-12
Arthur Sehn, Serge Autexier (Hrsg.)
Proceedings des Studentenprogramms der 18. Deutschen Jahrestagung für Künstliche Intelligenz KI-94
69 Seiten

D-94-11
F. Baader, M. Buchheit, M. A. Jeusfeld, W. Nutt (Eds.)
Working Notes of the KI'94 Workshop: KRDB'94 - Reasoning about Structured Objects: Knowledge Representation Meets Databases
65 pages

Note: This document is no longer available in printed form.

D-94-10
F. Baader, M. Lenzerini, W. Nutt, P. F. Patel-Schneider (Eds.)
Working Notes of the 1994 International Workshop on Description Logics
118 pages

Note: This document is available for a nominal charge of 25 DM (or 15 US-$).

D-94-09
Technical Staff
DFKI Wissenschaftlich-Technischer Jahresbericht 1993
145 Seiten

D-94-08
Harald Feibel
IGLOO 1.0 - Eine grafikunterstützte Beweisentwicklungsumgebung
58 Seiten

D-94-07
Claudia Wenzel, Rainer Hoch
Eine Übersicht über Information Retrieval (IR) und NLP-Verfahren zur Klassifikation von Texten
25 Seiten

D-94-06
Ulrich Buhrmann
Erstellung einer deklarativen Wissensbasis über recyclingrelevante Materialien
117 Seiten

D-94-04
Franz Schmalhofer, Ludger van Elst
Entwicklung von Expertensystemen: Prototypen, Tiefenmodellierung und kooperative Wissensevolution
22 Seiten

D-94-03
Franz Schmalhofer
Maschinelles Lernen: Eine kognitionswissenschaftliche Betrachtung
54 Seiten

Note: This document is no longer available in printed form.

D-94-02
Markus Steffens
Wissenserhebung und Analyse zum Entwicklungsprozeß eines Druckbehälters aus Faserverbundstoff
90 pages

D-94-01
Josua Boon (Ed.)
DFKI-Publications: The First Four Years 1990 - 1993
75 pages

Working Notes of the KI'96 Workshop on Agent-Oriented Programming and Distributed Systems

Klaus Fischer (Ed.)

D-96-06
Document