Dissertation zur Erlangung des Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
der Technischen Fakultät der Universität des Saarlandes

MASSIVE: Software Engineering for Multiagent Systems

vorgelegt von

Jürgen Lind

Saarbrücken, Dezember 1999

Gutachter
Prof. Dr. (Ph.D.) grad. Ing. J. Siekmann
Prof. Dr. J. Hertel

Vorsitzender des Prüfungsausschusses
Prof. Dr. W. Wahlster

Tag des Kolloqiums
17. Februar 2000

Zusammenfassung

In dieser Dissertation präsentieren wir ein pragmatisches Vorgehensmodell für die Entwicklung von Multiagentensystemen, das auf einer Kombination von Standardmethoden der Softwaretechnik aufgebaut ist. Die vorgestellte Methode besteht aus einem Produktmodell zur Beschreibung des Zielsystems, einem Prozeßmodell zur Konstruktion des Produktmodells und einem institutionellen Rahmen zur Unterstützung des projektübergreifenden Lernens. Die Methode ist während der letzten Jahre gereift und wurde bereits in verschiedenen industriellen Applikationen erfolgreich eingesetzt.

Kurzfassung

Um Agenten und Multiagentensysteme in der Industrie etablieren zu können ist es notwendig, existierende Softwaretechnik-Methoden weiter zu entwickeln und auf die speziellen Bedürfnisse dieser Klasse von Problemen zuzuschneiden. Darüberhinaus sind Softwaretechnik-Methoden für Agenten und Multiagentsysteme ein interessantes Forschungsgebiet, das neue Beschreibungsmechanismen für komplexe, dynamische Systeme hervorbringt. Wir stellen in dieser Dissertation eine Softwaretechnik-Methode für Multiagentensysteme vor, die sowohl den Forderungen aus der Industrie nach klar spezifizierten Prozeß- und Produktmodellen als auch akademischen Forderungen nach neuen Beschreibungsmodellen für diese Klasse von Systemen Rechnung trägt.

Die in der vorliegenden Arbeit präsentierte pragmatische Entwicklungsmethode orientiert sich an den speziellen Anforderungen von Multiagentensystemen als komplexe, dynamische Systeme. Die anforderungsgetriebene, iterative und sichten-orientierte MultiAgent-SystemS-Iterative-View Engineering-Methode (MASSIVE) kombiniert verschiedene Standardtechniken der Softwaretechnik wie z. B. Mehr-Sichten-Modellierung oder Roundtrip Engineering mit einer Erfahrungsdatenbank zum projektübergreifenden Lernen und zur systematischen Wiederverwendung. Genauer gesagt, stellt die MASSIVE Methode ein *multiagenten-spezifisches Produktmodell* zur Beschreibung der Entwurfsentscheidungen sowie der Implementierung des Systems, ein *Makro-Prozeßmodell* für den gesamten Lebenszyklus des Systems sowie *Mikro-Prozeßmodelle* für einzelne Sichten und einen *institutionellen Rahmen* für projektübergreifendes Lernen und systematische Wiederverwendung, der es ermöglicht, die Prozesse und Produkte eines speziellen Projekts anhand existierender Erfahrungen zu konfigurieren, zur Verfügung.

Das Kernkonzept der MASSIVE Methode ist ein generisches System von

Sichten, das es erlaubt, eine Vielzahl von speziellen Produktmodellen für einzelne Anwendungsgebiete von Multiagentensystemen zu spezifizieren. Eine Sicht repräsentiert dabei eine Menge von konzeptionell zusammengehörenden Eigenschaften des Zielsystems, d. h. eine Sicht ist eine Projektion des Gesamtmodells auf einen speziellen Aspekt des Systems. Idealerweise kann ein System in mehrere, unabhängige Teile mit wohldefinierten Schnittstellen zwischen einzelnen Teilen zerlegt werden wobei diese Schnittstellen explizit repräsentiert werden, um bestehende Abhängigkeiten klar zu dokumentieren. Das in der vorliegenden Arbeit vorgeschlagene System besteht aus sieben Sichten.

Die *Aufgaben-Sicht (Task view)* beschreibt das System unter rein funktionalen Gesichtspunkten ohne bereits Bezug auf bestimmte Lösungskonzepte zu nehmen; die *Umgebungs-Sicht (Environment view)* enthält die Beschreibungen der Systemumgebung aus der Sicht des Entwicklers und aus der Sicht des Systems selbst; die *Rollen-Sicht (Role view)* analysiert die funktionalen und physikalischen Abhängigkeiten bzw. die daraus möglichen Rollenmodelle für Agenten; die *Interaktions-Sicht (Interaction view)* beschreibt die zur Problemlösung notwendigen Interaktionen zwischen Agenten in Form von prototypischen Interaktions-Schemata; die *Gesellschafts-Sicht (Society view)* modelliert die Struktur der Agentengesellschaft; die *Architektur-Sicht (Architectural view)* enthält die softwaretechnische Modellierung des Zielsystems bzw. seiner Komponenten und Letztendlich deckt die *System-Sicht (System view)* alle Aspekte des Systems ab, die sich nicht einer einzelnen der anderen Sichten zuordnen lassen.

Die Grundidee des vorgeschlagenen Prozeßmodells, welches zur Konstruktion des im vorigen Abschnitt skizzierten Produktmodells herangezogen werden kann, basiert auf einer iterativen Verfeinerung der einzelnen Sichten bis hin zur Implementierung. Das Gesamtmodell setzt sich dabei aus mehreren Teilmodellen für die einzelnen Sichten zusammen. Die wesentlichen Vorteile des Prozeßmodells sind die Fähigkeit, mit unvollständigen Informationen zu arbeiten und die sich aus den frühzeitigen Schritten hin zur Implementierung ergebenden Möglichkeiten der Risiko-Minimierung.

Die vorgeschlagenen Prozeßmodelle sowie das oben vorgestellte Produktmodell sind keineswegs vollständig und unveränderlich festgeschrieben. Aufgrund der wachsenden Erfahrung im Lauf der Softwareenwicklungsaktivitäten sollten diese Modelle an die spezifischen Gegebenheiten einer Organisation angepaßt und ständig verbessert werden. Aus diesem Grund ist die Projektabwicklung in einen größeren organisatorischen Rahmen eingebettet, der systematisches Lernen und Wiederverwendung von Wissen ermöglicht. Diese Erfahrungsfabrik stellt die für systematische Verbesserung notwendigen Strukturen und organisatorischen Rahmenbedingungen zur Verfügung und ermöglicht so den Aufbau einer mit der Organisation wachsenden Erfahrungsdatenbank.

Die in der vorliegenden Arbeit vorgestellte MASSIVE Methode wurde im Lauf der letzten Jahre entwickelt und verfeinert und zur Entwicklung von sehr verschiedener Multiagenten-Anwendungen, z. B. in der Kostenoptimierung von Gütertransporten, in der intermodalen Routenplanung und in einer Bibliothek zur Unterstützung von verteilter Suche in sehr großen Suchräumen, erfolgreich eingesetzt.

Abstract

MASSIVE is a pragmatic development model for multiagent systems based on a combination of standard software engineering techniques. It features a product model to describe the target system, a process model to construct the product model and an institutional framework that supports learning and reuse over project boundaries. The method has evolved over several years and it has been successfully applied and refined in various industrial research projects.

Summary

The advancement of engineering methods for Agents and multiagent systems are incresingly used in the software industry, but existing software development methods are often not suited for such highly complex and dynamic systems. Thus, new methods are needed in order to establish this technology in an industrial context. Furthermore, engineering agents and multiagent systems is an interesting research field that can provide new methods and techniques for a better understanding and modeling of these applications.

It is the goal of this dissertation to provide a development method for multiagent systems that satisfies both industrial needs for clearly specified product and process models that can be readily used by software engineers by providing a better understanding and modeling of multiagent systems.

We propose a pragmatic method for the development of multiagent systems that is requirements-driven, view-oriented and iterative. The MultiAgent SystemS Iterative View Engineering (MASSIVE) development method provides a *multiagent system specific product model* that is used to describe the design and the implementation of the target system, a *macro process model* that covers the entire life cycle of the system development as well as several *micro process models* that are used to describe particular aspects of the target system. An *institutional framework* supports learning and reuse over project boundaries and allows the project management to configure the process and product models of a particular project.

The MASSIVE method consists of three major parts. The core of the method is a system of views that form the conceptual basis for a wide range of product models that are developed and refined throughout various software projects. A view represents a set of conceptually linked features of the target system, ie. a view is a projection of the complete model onto a particular subject. Ideally, the target system can be decomposed into several independent units with well defined interfaces that are explicitly modeled in order to clarify the interdependencies that exist within the model. In the MASSIVE method, we propose a sys-

tem that consists of seven views.

The *Task view* models the system task from a solely functional point-of-view without a commitment to a particular technology or system design structure; the *Environment view* describes the system environment from the point-of-view of the system designer and from the point of view of the system; the *Role view* analyzes the functional and physical decompositions of the Task view and models the roles that occur within the target system; the *Interaction view* defines what forms of interactions can take place in the system and models the interaction protocols for the participating agents; the *Society view* provides the structure of the agent society within the target system; the *Architectural view* models the system architecture in the context of a given environment and the *System view*, finally, contains all aspects of the target system that affect the system as a whole or that cannot be assigned to any of the other views.

The main idea of the suggested process model is an Iterative View Engineering approach that is itself based on Iterative Enhancement and Round-trip Engineering. The overall process model contains itself several small micro models that are used for individual tasks within the design process for a particular view. The advantages of this life-cycle model are that it can deal with an incomplete problem specification because of the Iterative Enhancement approach. The model is also useful for early risk detection because parts of the model are incrementally implemented and can direct the project managers view to critical regions.

The micro models as well as the overall process model are in not fixed for the entire lifetime of the project model. Just like the views discussed earlier, they are also subject to changes and refinements during the course of time. In order to preserve these adaptations and to make them accessible to others, the process model and the product model are both embedded into a larger organizational structure called the *Experience Factory*. The Experience Factory provides the formal framework for a permanent learning process over project boundaries that eventually captures the multiagent experience of organization in terms of specific product and process models for various domains.

The MASSIVE method that is presented in this dissertation has evolved over several years and it has been successfully applied and refined in different types of multiagent systems among others for cost optimization in railroad transport, intermodal route planning and an agent-based library for distributed search applications.

MASSIVE: Software Engineering for Multiagent Systems

Jürgen Lind
German Research Center for AI (DFKI)
Im Stadtwald B36
D-66123 Saarbrücken, Germany

lind@dfki.de

For my parents,
my grandparents
and in memory of Walter Becker.

Contents

1 Introduction **1**

1.1 Where are we? . . . 2
1.2 What do we want? . . . 4
1.3 What do we provide? . . . 6
1.4 Case Studies . . . 10
1.5 Acknowledgments . . . 11

2 An Introduction to Multiagent Systems **13**

2.1 Intelligent Agents . . . 13
2.1.1 What's an Agent, anyway? . . . 14
2.1.2 Roles . . . 16
2.1.3 Agent Architectures . . . 18
2.1.4 Agents, Roles and Architectures . . . 19
2.2 Systems of Agents . . . 21
2.2.1 Interaction . . . 21
2.2.2 The Social Dimension . . . 22
2.3 Summary . . . 24

3 Basic Concepts in Software Engineering **25**

3.1 Aspect-Oriented Programming . . . 25
3.2 Process Models . . . 27
3.2.1 Waterfall Model . . . 28
3.2.2 V-Model . . . 30
3.2.3 Cleanroom . . . 33
3.2.4 Prototyping . . . 35
3.2.5 DSDM . . . 36
3.2.6 Spiral Model . . . 38
3.3 Quality Management and Systematic Learning . . . 40
3.3.1 The Quality Improvement Paradigm . . . 41
3.3.2 Experience Factory . . . 42

3.4 The Unified Modeling Language 45
3.5 Summary 47

4 The Conceptual Framework of MASSIVE **49**
4.1 Features 49
4.2 Views 52
4.2.1 What and Why? 52
4.2.2 A View System for Multiagent Systems 53
4.3 Iterative View Engineering 61
4.4 Putting It All Together 64
4.5 Summary 67

5 MASSIVE **Views** **69**
5.1 A Brief Introduction to Train Coupling- and Sharing (TCS) 71
5.2 Task View 73
5.2.1 General 73
5.2.1.1 Functional Requirements 74
5.2.1.2 Nonfunctional Requirements 76
5.2.1.3 Basic Problem Solving Capabilities 76
5.2.2 Case Study 77
5.2.2.1 Functional Requirements 77
5.2.2.2 Nonfunctional Requirements 81
5.2.2.3 Basic Problem Solving Capabilities 81
5.3 Environment View 88
5.3.1 General 88
5.3.1.1 Developers Perspective 88
5.3.1.2 Systems Perspective 91
5.3.2 Case Study 92
5.3.2.1 Developers Perspective 92
5.3.2.2 Systems Perspective 93
5.4 Role View 93
5.4.1 General 93
5.4.1.1 Role Definition 94
5.4.1.2 Role Assignment 96
5.4.2 Case Study 97
5.4.2.1 Functional Grouping 97
5.4.2.2 Physical Grouping 98
5.4.2.3 Aggregation 98
5.4.2.4 Role assignment 99
5.5 Interaction View 99

5.5.1 General . . . 99
5.5.1.1 Conflict Resolution . . . 100
5.5.1.2 Protoz: A Protocol Compiler . . . 102
5.5.1.3 Interaction Design . . . 105
5.5.2 Case Study . . . 110
5.5.2.1 Contract-Net . . . 110
5.5.2.2 Simulated Trading . . . 112
5.5.2.3 Client Server . . . 117
5.6 Society View . . . 117
5.6.1 General . . . 117
5.6.1.1 Characterization of Social Systems . . . 118
5.6.1.2 Designing Social Systems . . . 120
5.6.2 Case Study . . . 121
5.6.2.1 Dynamic Clustering . . . 123
5.6.2.2 Static Clustering . . . 123
5.7 Architectural View . . . 125
5.7.1 General . . . 125
5.7.1.1 The Architectural Feature Space . . . 126
5.7.1.2 System Architecture . . . 129
5.7.1.3 Agent Architecture . . . 132
5.7.2 Case Study . . . 135
5.7.2.1 System Architecture . . . 135
5.7.2.2 Agent Architecture . . . 139
5.8 System View . . . 144
5.8.1 General . . . 144
5.8.1.1 User Interface Design . . . 144
5.8.1.2 Exception Handling . . . 147
5.8.1.3 Performance Engineering . . . 148
5.8.1.4 Deployment . . . 152
5.8.1.5 Summary . . . 155
5.8.2 Case Study . . . 156
5.8.2.1 Graphical User Interface . . . 156
5.8.2.2 Exception Handling . . . 157
5.8.2.3 Performance Engineering . . . 158
5.8.2.4 Deployment . . . 160
5.9 Summary . . . 160

6 Further Case Studies 163
6.1 Personal Travel Assistant: Intermodal Route Planning . . . 163
6.1.1 Task View . . . 164

6.1.2 Environment View 171
6.1.3 Role View 172
6.1.3.1 Functional Roles 172
6.1.3.2 Aggregation 173
6.1.4 Interaction View 173
6.1.5 Society View 177
6.1.6 Architectural View 178
6.1.6.1 System Architecture 178
6.1.6.2 Agent Architecture 179
6.1.7 System View 181
6.1.7.1 Graphical User Interface 181
6.1.7.2 Performance Engineering 182
6.1.8 Summary 189
6.2 The TEAMWORK LIBRARY 190
6.2.1 Task View 190
6.2.2 Environment View 192
6.2.3 Role View 193
6.2.3.1 Functional Grouping 193
6.2.3.2 Physical Grouping 195
6.2.3.3 Aggregation 195
6.2.4 Interaction View 195
6.2.5 Society View 199
6.2.6 Architectural View 200
6.2.6.1 System Architecture 200
6.2.6.2 Agent Architecture 201
6.2.7 System View 203
6.2.7.1 User Interface 203
6.2.7.2 Performance Engineering 204
6.2.7.3 Deployment 206
6.3 Summary 207

7 Related Work 209
7.1 Development Methods for Multiagent System 209
7.1.1 Burmeister 210
7.1.2 Kinny and Georgeff 213
7.1.3 DESIRE 214
7.1.4 MAS-CommonKADS 216
7.1.5 Wooldridge, Jennings and Kinny 220
7.2 Toolkits for Agent-Based Applications 224
7.2.1 SIF 224

7.2.2 ZEUS 228
7.2.3 Swarm 231
7.3 Summary 232

8 Conclusion **233**
8.1 Achievements 233
8.2 Further Work 235

A Glossary **239**

B Example Protoz Specification for the Contract-Net Protocol **243**

Bibliography **247**

List of Figures

1.1 Context . . . 2
1.2 The Goal Method . . . 5
1.3 Contents Roadmap . . . 9

2.1 Perceive-Reason-Act Cycle . . . 14
2.2 A Single Agent . . . 15
2.3 A Container Stock Area . . . 16
2.4 Agents and Roles . . . 19
2.5 A Generic Agent Application Architecture . . . 20
2.6 Sociogram . . . 23

3.1 Aspect Weaver . . . 26
3.2 Ideal Software Development Process . . . 27
3.3 Waterfall Model . . . 29
3.4 Product-States . . . 30
3.5 V-Model . . . 31
3.6 Cleanroom Model (DYER, 1992) . . . 33
3.7 Classes of Prototypes (BALZERT, 1998A) . . . 36
3.8 DSDM (THE DSDM CONSORTIUM, 1998) . . . 38
3.9 Spiral Model (BOEHM, 1988) . . . 39
3.10 The Quality Improvement Paradigm (BASILI ET AL., 1994) . . . 42
3.11 The Experience Factory (BASILI ET AL., 1994) . . . 44

4.1 Connections between Model and Implementation . . . 50
4.2 Feature refinement . . . 51
4.3 A Generic View System (BALZERT, 1998B) . . . 53
4.4 Features and Views . . . 54
4.5 Shared Features and Feature Refinement . . . 54
4.6 MASSIVE Views . . . 57
4.7 The View System Tree . . . 60
4.8 Round-trip Engineering . . . 61
4.9 Iterative Enhancement . . . 62

4.10 Micro Processes 63
4.11 Iterative View engineering 64
4.12 The MASSIVE Method Model 65
4.13 The MASSIVE Method as UML Diagram 65

5.1 Views and Multiagent Systems 70
5.2 Hierarchical freight haulage 72
5.3 Train Coupling and Sharing (TCS) 73
5.4 Example for a Task Tree 75
5.5 TCS Simulator Workflow (UML) 78
5.6 Example railroad network 78
5.7 Task Tree of the TCS Domain 80
5.8 Time Window Propagation 82
5.9 Planer (UML) 83
5.10 Action Scheduling 86
5.11 Perspectives on the Environment 88
5.12 Hybrid Role Identification 95
5.13 Information Exchange 106
5.14 Task Decomposition 107
5.15 Centralized Market 108
5.16 Distributed Market 108
5.17 Generic Blackboard Architecture 109
5.18 Contract-Net Protocol 111
5.19 Contract-Net Protocol (UML) 113
5.20 Contract-Net Manager 114
5.21 Contract-Net Bidder 115
5.22 Simulated Trading 115
5.23 Simulated Trading (UML) 116
5.24 Example for a hierarchical society 119
5.25 Social System Performance (adapted from (SO AND DURFEE, 1998)) 121
5.26 Clustering 124
5.27 Architectural Design Space 129
5.28 Generic Software Architectures 130
5.29 System Architecture 135
5.30 System Architecture (UML) 136
5.31 Message Passing in TCS/MAS 138
5.32 Message Passing in TCS/MAS 139
5.33 InteRRap 141
5.34 Union Agents 142
5.35 Union Agents (UML) 143

5.36 The Performance Engineering Process 149
5.37 The Performance Improvement Cycle 151
5.38 Deployment Life cycle 153
5.39 Elements of the TCS/MAS GUI 156
5.40 The MASSIVE View System 161

6.1 IMRP Example 164
6.2 IMRP Broker Architecture 174
6.3 Distributed IMRP Architecture 176
6.4 Basic Society Structure 177
6.5 Hierarchical IMRP Organization 178
6.6 MECCA Agent Architecture 181
6.7 IMRP Graphical User Interface 182
6.8 Generic Selection Function 184
6.9 Example 1 187
6.10 Example 2 187
6.11 Example 3 188
6.12 Team Meeting 196
6.13 Broadcast Protocol 198
6.14 Team Structure 200
6.15 Teamwork Agents 202
6.16 Broadcast Statistic Tool 204

7.1 The Generic DESIRE Agent Architecture 216
7.2 Hierarchical Set of Models 218
7.3 Model Relations (WOOLDRIDGE ET AL., 1999) 221
7.4 The EMS Idea 225
7.5 The EMS Model 225
7.6 Conceptual Agent Model of SIF 226
7.7 Information and Control Flow in SIF 227
7.8 SIF User Interaction 228
7.9 ZEUS Agent Architecture (NWANA ET AL., 1999) 229
7.10 ZEUS Screenshot 230
7.11 Swarm Hierarchy 231

8.1 MADS View System Selection Process 236
8.2 MADS Supported View Construction 236

Process Models

1	Cleanroom	34
2	Quality Improvement Paradigm	43
3	MASSIVE	66
4	MASSIVE Role Modeling	96
5	Protoz Protocol Design	104
6	MASSIVE Society Design	122
7	MASSIVE User Interface Design	146
8	Performance Engineering	150
9	System Deployment	155
10	Burmeister	212
11	Kinny and Georgeff	214
12	DESIRE	217
13	MAS-CommonKADS	219
14	Wooldridge, Jennings and Kinny	223

Our ability to imagine complex applications will always exceed our ability to develop them.

Grady Booch

CHAPTER 1

Introduction

IT IS A WIDELY AGREED FACT in the Software Engineering Community that any software life cycle, process or product model must be tailored towards the characteristic needs of the application domain of the target system (BASILI, 1989), (BASILI ET AL., 1994) as there is no "silver bullet" (BROOKS, 1986), ie. a method that can be used for all types of problems.

The strong need for agents and agent systems development methods in industry is exemplified by the major European telecommunication companies that have launched a joint research project to foster the definition of a development method for agent applications (EURESCOM, 1999). Furthermore, the engineering topic of agents and multiagent systems is not only a technical matter that has been picked up by industry, it is also an interesting research field that can provide new methods and techniques for a better understanding and modeling of these highly complex systems (WOOLDRIDGE, 1997), (JENNINGS ET AL., 1998), (JENNINGS, 1999).

However, development methods for agent-based and multiagent systems are still in their infancy and must be more advanced in order to establish the technology in an industrial context. As stated in (PARUNAK, 1999): "Relatively less attention has been paid to the important question of the process that designers go through. Industrial users will use agents more readily if basic principles and guidelines are available ...".

It is the goal of this dissertation to provide a development method for multiagent systems that provides a better understanding and modeling of multiagent

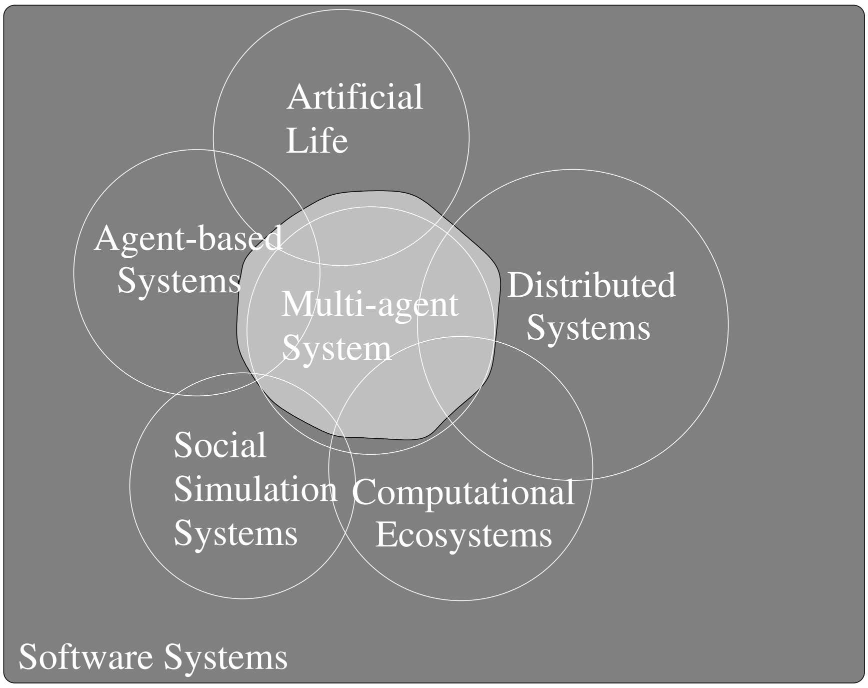


Figure 1.1: Context

system. It is a step towards industrial needs for clearly specified product and process models that can be readily used by software engineers.

In the rest of this introduction, we will outline the context of the method more precisely by relating the type of systems that can be developed using the method to other fields of computer science. After the definition of the scope of the method, we will then describe what an ideal method should provide to the user. Then we will outline the method that is described in this dissertation and give an overview over the structure of the remaining chapters.

1.1 Where are we?

Probably the hardest problem in designing a development method for multi-agent systems is to define the scope of the method. It should neither be too specific in that it covers only a small fraction of multiagent applications, nor should it be too general because unrelated details make the method less useful in the specific context.

Figure 1.1 shows the connection between multiagent applications and applications in related fields of computer science. The field of multiagent applications has basically two roots: distributed systems and agent-based computing. Distributed systems (FOX, 1981), (MULLENDER, 1993) (TANENBAUM, 1988) is the sub-field of computer science that is concerned with the design and implementation of computer applications where several computers or processors cooperate in some way (TEL, 1994). Distributed systems are often systems where the participating entities have little or no autonomy in what they do. The designer specifies the entire system behavior and the only purpose for the distribution is usually performance enhancement through the exploitation of parallelism. Agent-based computing (SHOHAM, 1993), (BRADSHAW, 1997), (HUHNS AND SINGH, 1998) (JENNINGS AND WOOLDRIDGE, 1998), on the other hand, is concerned with the design and implementation of flexible, autonomous entities (JENNINGS ET AL., 1998). These entities – the agents – are situated in an environment that they perceive with their sensoric apparatus and on which they can act upon using their effectors. The agents are equipped with reasoning capabilities that allow them to deliberate upon their next actions in order to fulfill their goals.

Merging the two fields together provides a rather good characterization of the basic structure of multiagent systems. Although there is no generally agreed definition for the term "multiagent system" the different characterizations for multiagent systems that have been proposed, eg. (GASSER, 1995), (O'HARE AND JENNINGS, 1996) (GOODWIN, 1993), (FULBRIGHT AND STEPHENS, 1994) (FRANKLIN AND GRAESSER, 1997) and (WOOLDRIDGE AND JENNINGS, 1998) share the basic ideas and assumptions. A definition that complies to most of the existing characterizations is given below.

DEFINITION 1.1.1 (MULTIAGENT SYSTEM) **Multiagent systems** *are systems with a variable number of interacting, autonomous entities that communicate with each other using flexible, complex protocols. The agents within a multiagent system usually have complex individual components and act concurrently in a distributed environment.*

Agents within a multiagent system do not necessarily cooperate, systems with *competing* agents also exist. This relates multiagent systems to computational ecosystems (KEPHART ET AL., 1989), (PARUNAK ET AL., 1997). In such systems, the designer tries to imitate the situation within a natural ecosystem where the entities have strong interdependencies that can have positive or negative effects. Thus the entities do not only cooperate, they also compete for resources if it is necessary for them to improve their performance.

There are two other related fields that have only minor connections to multiagent systems, one is artificial life and the other are social simulation systems. Research in artificial life (LANGTON, 1989), (BODEN, 1996), (ADAMI, 1998) is concerned with the formal basis of life and the mechanisms that produce life-like behavior (FRANKLIN, 1997). These mechanisms are then implemented in computer programs that synthesize the behaviors and allow for the analysis of the emergent functionality. Social simulation systems (HEISE, 1992), (EPSTEIN AND AXTELL, 1996), finally, were introduced by sociologists to simulate the behavior of large groups of individuals. These systems are useful because they allow the researcher to abstract away from individual characteristics of humans and to focus on particular properties of the system. However, these systems only play a minor role in the field of computer science.

We will now define the intended coverage of a design method for multiagent systems more precisely as indicated in Figure 1.1 by the lightly grey shaded area. The method should approximate the circle that represents the multiagent applications as close as possible in order to achieve a maximum degree of coverage.

1.2 What do we want?

The above definition of the term "multiagent system" already suggests that applications that are built on this concept are far from being easy to design. More specifically, multiagent systems are usually *complex*, *decentralized* systems that are often *ill-structured* but on the other hand have the desirable properties of being *modular* and *changeable* (PARUNAK, 1999). Thus we want to define a general software development method according to the following definition that supports the development of systems with these specific properties.

DEFINITION 1.2.1 (SOFTWARE DEVELOPMENT METHOD (BOOCH, 1996)) *A* **software development method** *encompasses a* notation*, whose purpose is to provide a common means of expressing strategic and tactical decisions, ultimately manifesting themselves in a variety of artifacts and a* process*, responsible for specifying how and when certain artifacts should be produced.*

DEFINITION 1.2.2 (NOTATION (BOOCH, 1996)) *A* **notation** *serves as the* language for communicating *decisions that are not obvious or cannot be inferred from the code itself,* provides rich enough semantics *sufficient to capture all important strategic and tactical decisions and offers a* concrete form for humans to reason *about decisions.*

The basic structure of an ideal development method as seen by the author is shown in Figure 1.2. According to the figure and to the prior definitions, the

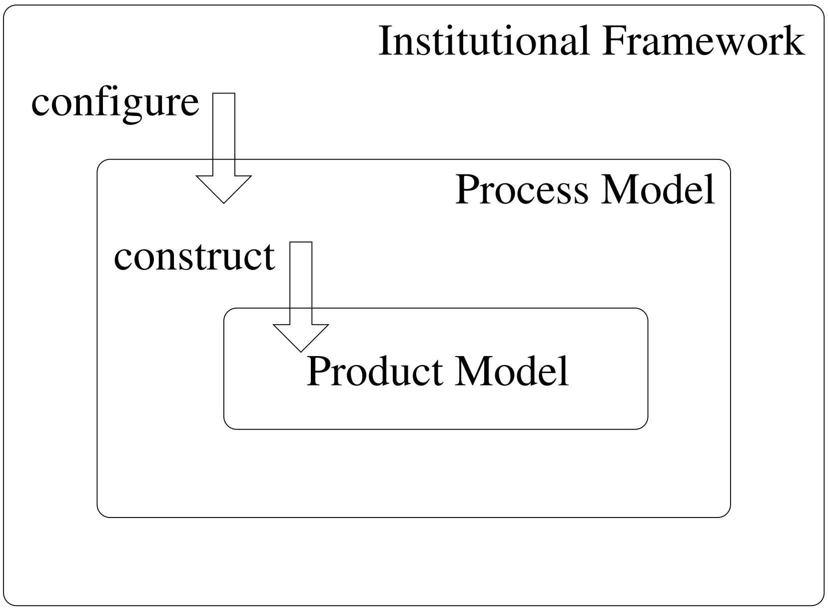


Figure 1.2: The Goal Method

ideal method provides a notation in form of one or more *Product models* that are used to describe the design and/or the implementation of a particular system, one or more *Process models* that guide the developer in constructing the product models according to the user requirements and finally, the ideal method provides an *Institutional framework* that tells the project management how the development process must be configured to yield the best possible result.

Furthermore, the ideal development method should exhibit some additional properties such that it can be used for a broad range of multiagent application projects.

Now, what does this mean? Firstly, a *flexible* method is not committed to a particular technology, eg. to a specific platform, agent technology or agent architecture such as BDI agents (RAO AND GEORGEFF, 1995) or a certain software development process model. Then, it should be possible to *adapt* the method to the particular needs of the organization that is using it and to the specific project that it is used for. Furthermore,the method should be *simple* in that it is straightforward to use and does not require extensive training for the user of the method. The simpler it is, the more will it be accepted in the multiagent systems community and a good means to achieve this goal is to provide a set of core ideas that can be used directly and to reserve more complicated features of the method to the experienced user. To be *open*, the method must be applicable in a wide variety of development environments. For example, it should not be limited to a particular programming paradigm (eg. object-oriented programming) or even a particular programming language (eg. Java). Instead, the method must allow the user to maintain his or her development environment as far as possible and it must be changeable according to specific user requirements. Furthermore, the method must be *scalable* in order to be applicable to development problems of almost any size, ranging from small, single person projects up to large, industrial projects. An important aspect that must be considered in the development of a design method is that it should enable the designer or – even better – the organization as a whole to *represent and preserve any kind of knowledge* that was obtained during the project execution and to *transfer this knowledge* over project boundaries in order to support knowledge reuse. An overemphasis on originality can easily lead to the re-invention of the wheel and this is likely to repeat mistakes that were already made by other designers and could be avoided (PARNAS, 1996). Ideally, the institutional framework that is needed to establish such a project framework should be small and allow the organization to introduce the scheme with as few impact on the ongoing business processes as possible. The last requirement, finally, is that the design method for multiagent system should be *clearly related to existing Software Engineering approaches.* Thus, we do not want a method that operates in "free space" without making use of the benefits of software engineering research of the last 30 years. Although this requirement seems pretty obvious, it cannot always be taken for granted as shown in (WOOLDRIDGE ET AL., 1999) where the authors propose a design method that operates "on-top of standard SE techniques". Instead, we want a method that operates *within* a standard software engineering context.

1.3 What do we provide?

We propose a pragmatic method for the development of multiagent systems that can account for the specific requirements mentioned above. The basic idea of the method is outlined in Figure 1.2. We have combined several standard software engineering techniques such as Multi-view Modeling (VERLAGE, 1994) and Round-trip Engineering (BALZERT, 1998A) and focused them on the special context of multiagent systems. Furthermore, we also propose an institutional framework that supports learning over project boundaries.

The resulting MultiAgent SystemS Iterative View Engineering (MASSIVE) development method continuous previous work of the author (LIND, 1999A). However, we do not attempt to invent a uniform method for all types of applications (the "silver bullet") but move modestly making our experience in the design of multiagent systems available to other system designers.

The most important characteristics that distinguishes the approach presented in this dissertation from other approaches are listed below.

Requirements-driven vs. technology-driven In a requirements-driven development approach, the requirements of the target system determine the technology that is to be used for the system design and implementation and not vice-versa. Before the designer commits to a particular technology, however, it is necessary to analyze whether the intended technology is appropriate for the particular problem or not. This is clearly pointed out in (WOOLDRIDGE AND JENNINGS, 1998) and it should always be kept in mind that "if the only thing you got is a hammer, everything looks like a nail". Thus, the designer must not limit himself to a particular technology if it is not appropriate to the problem under consideration and consequently, a requirements-driven approach starts earlier in the life cycle of the target system when the fundamental characteristics of the problem under consideration are analyzed.

In a rigorous requirements-driven approach, a functional analysis leads to a task hierarchy that will be built without a particular technology in mind. After the task hierarchy is constructed, objective decision functions are then applied to it in order to find the most appropriate technologies. However, the basic difficulty of this idea is that until now, no such decision functions exist and thus is subject to further research in the multiagent community (JENNINGS ET AL., 1998).

View-oriented vs. model-oriented All of the related approaches discussed in Section 7.1 develop several distinct models of the target system that must be subsequently integrated in order to construct the final system. This integration process is not always trivial, as the links between the models are usually neither explicitly defined nor obvious. Thus, we suggest adopting a view-oriented approach instead. A view-oriented method deals with the system as a whole and uses different perspectives on the entire system as fundamental abstraction. The system is not decomposed but rather viewed from different angles with different foci on particular aspects. The advantage of this approach is that it avoids the integration process because the model is always consistent from any view as changes in one view are always directly propagated to other views.

Iterative vs. sequential A sequential process model has the optimistic assumption, that everything will work out as desired and that all documents are complete, correct and consistent once they are created. This assumption may hold in well-understood problem domains for which a lot of systems have been constructed, but it is unlikely that it holds in the case of multiagent systems because this class of systems is not well-understood at all until now. Any process model that is to be used in this context should therefore reflect the fact that the system requirements and design documents are normally incomplete, incorrect and inconsistent due to the limited experience with this class of systems. To overcome these difficulties, it is a good idea to iterate the processes that are applied to construct the documents and to evaluate the results of each iteration according the explicit quality measures that are defined early in the development process. Although this approach does not guarantee that the documents capture all relevant aspects in the correct way, it nonetheless reduces the risk of errors because of the repeated quality evaluations.

Balanced vs. biased Traditional Software Engineering methods are often biased towards the model aspects of the problem in question and neglect the code part of the resulting system. Novel trends in the Software Engineering community and especially the Object-Oriented part thereof, feature a point-of-view that treats the model and the code that implements the model as equally important parts of the whole (OPDYKE, 1992), (FOWLER, 1999), (BECK, 1999). In this dissertation, we will adopt this point-of-view and present a development method that clearly relates the parts of the model and the matching parts of the code and that supports the construction of code that is well-structured and thus easy to maintain.

In particular, the MASSIVE method provides a *specific multiagent system product model*, a nested process model that consists of a *macro process model* that covers the entire life cycle of the system development as well as several *micro process models* that are used to describe particular aspects of the target system and a *framework for institutional learning* in the development of multiagent systems.

Besides these major contributions, the work on this dissertation has lead to some basic design decisions of the *Social Interaction Framework*, a Java-based development tool-kit and generic test bed for multiagent applications that will be described briefly in Section 7.2.1 and to the development of a protocol specification system that provides a protocol language and a compiler for the translation of an abstract protocol definition to Oz (PROGRAMMING SYSTEMS LAB, 1999) code. The PROTOZ system will be presented in Section 5.5.1.2.

The best way to read this dissertation is shown in Figure 1.3 where we have depicted the main parts of the dissertation and their interconnections. In chap-

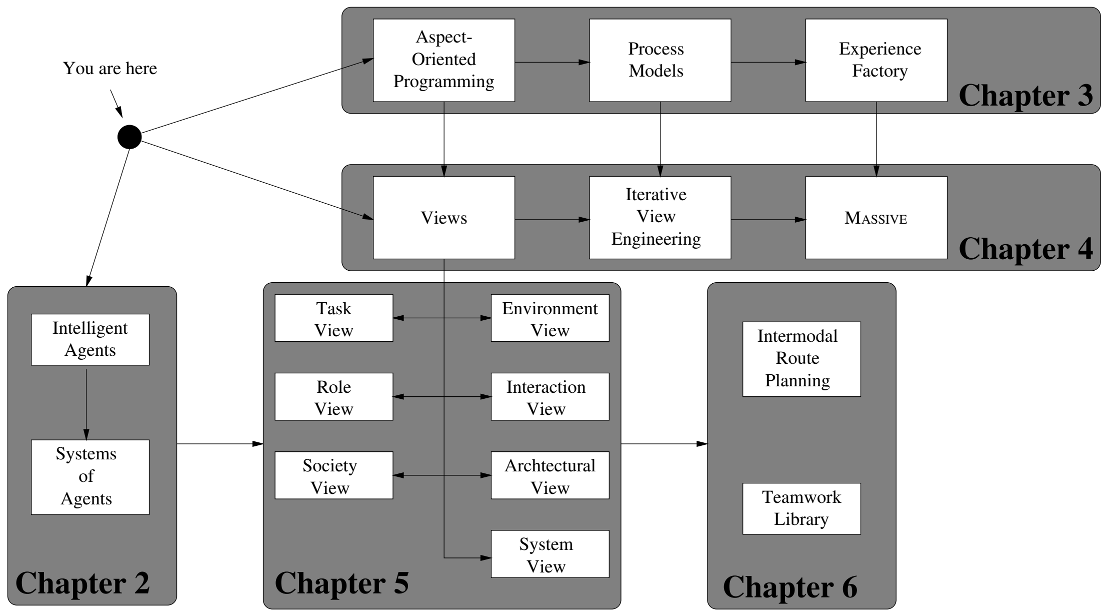


Figure 1.3: Contents Roadmap

ter 2 and chapter 3, we will provide the reader with some background information about Intelligent Agents and Multiagent Systems as well as the basic Software Engineering ideas that have made their way into the MASSIVE method. Chapter 4 will then discuss how the general Software Engineering concepts are mapped onto a particular instantiation and integrated into the overall context of MASSIVE. The most fundamental concept of the MASSIVE method is presented in depth in Chapter 5. Each section of this chapter is organized in two-fold manner; in the first part, general properties of multiagent systems with respect to a particular aspect are discussed. In the second part, the results of the first part are applied to a case study to demonstrate how the theoretical concepts are used in a real project. Additional case studies in which the MASSIVE method has been used are presented in Chapter 6. Not shown in Figure 1.3 is Chapter 7 that provides a review of related work on development methods for multiagent applications and Chapter 8 that concludes the paper and gives some outlook on further work.

1.4 Case Studies

The approach presented in this dissertation has evolved over several years and it has been successfully applied and refined in different types of multiagent systems:

- The multiagent solution for the Train Coupling- and Sharing (TCS) approach (VOGES AND MIERAU, 1997), (LIND AND FISCHER, 1998), (LIND AND BÖCKER, 1999), (LIND ET AL., 1999B), (LIND ET AL., 1999A) is a system for scheduling and cost optimization of a large number of railroad transportation tasks using novel railroad technologies. Multiagent systems are particularly well suited for the transportation domain because of the inherent complexity and natural degree of distribution (FISCHER ET AL., 1993) and a lot of research has be done in recent years (FISCHER ET AL., 1994), (FISCHER AND MÜLLER, 1995), (KUHN ET AL., 1994) to provide the necessary scientific background for further applications.
- The MOTIV/PTA (BAYRISCHE LANDESREGIERUNG, 1996), (SIEMENS AG, 1997) is a project concerned with the design of a **P**ersonal **T**rip **A**ssitant (PTA) for individual travelers according to the FIPA (FIPA, 1997) (FIPA, 1998) standards. One of the services required in the PTA domain is intermodal route planning for which we have implemented a multiagent solution.
- The TEAMWORK LIBRARY (DENZINGER, 1994), (LIND, 1996A), (DENZINGER AND LIND, 1996) is a framework for distributed search that was originally developed for equational theorem proving but that was also used in other application fields (KÖGL, 1995), (LEOPOLD, 1995).

In the course of this dissertation, we will frequently use the TCS project to illustrate the proposed method and how it is applied in a real-world application. The other projects are reviewed in an separate chapter at the end of the dissertation.

1.5 Acknowledgments

I would like to thank my supervisor Jörg Siekmann for stimulating my enthusiasm for Artificial Intelligence and for guiding me through the rough ways of writing a thesis. His immense knowledge and experience were a steady source of inspiration and our fruitful discussions helped my to bring my ideas into the present form.

I am also grateful to Joachim Hertel as the second supervisor of my thesis for teaching me a more practical view onto the software development process and the problems within an industrial environment.

Furthermore, I thank the people at the Multiagent systems research group of the DFKI: my project leader Klaus Fischer, our senior researchers Hans-Jürgen Bürckert and Matthias Klusch and my colleagues Petra Funk, Steve Allen, Thorsten Bohnenberger, Alastair Burt, Christian Gerber, Christoph Jung, Christian Russ and Gero Vierke. Some of the students there have also contributed to this thesis, especially Michael Rovatsos, Stefan Philipps and Michael Schillo. I would also like to thank the Omegas, the VSE people and the staff at the AGS for the pleasant working atmosphere during the last three years.

Christian Schulte and Ralf Scheidhauer from the Programming Systems Lab deserve some special acknowledgments for teaching me the bright sides of Oz.

In several industrial and research projects within the last three years, I worked with many people from different areas. In the TCS project, these were Jörg Böckers from the IVE and Bernd Zirkler, Wolfgang Voges and Ulrich Mierau from the Deutsche Bahn AG. The MoTiV-PTA project was a collaboration between the DFKI and Siemens AG where I had the pleasure to work with Donald Steiner, Hartmut Dieterich, Berhard Bauer and Gerd Völksen. The Teamwork library, finally, was built during my time at the University of Kaiserslautern together with Jörg Denzinger and others at the department for Rewriting Systems.

During a short term research fellowship, I had the opportunity to discuss aspects of my work with the some people at BT's Adastral Park Labs. Paul Kearney, Divine Ndumu, Brian Odgers, Simon Thompson, Matt Sullivan, Jaron Collins and Paul O'Brien helped my a great deal by pointing out matters that needed further clarification.

I am indebted to my friend and colleague Alexander Knecht from sd&m for the countless discussions, especially about "agents vs. objects" but also for the philosophical explorations of the essence of programming.

Finally, I thank my parents and the rest of my family for their support throughout my entire career and for keeping me on track when I was in danger of loosing it.

Nina – my love, my queen –, thank you for everything!

CHAPTER 2

An Introduction to Multiagent Systems

THE SCIENTIFIC ROOTS OF MULTIAGENT SYSTEMS (MAS) RESEARCH have been outlined in the Introduction; in this chapter, we will extend the introductory remarks and provide the reader with a somewhat more detailed view. We will start with an introduction to intelligent agents and then extend the single-agent case to systems with several intelligent agents. The basic concepts are still presented from a very general point-of-view and will be refined in subsequent chapters; the reader already familiar with with MAS may safely skip this chapter and continue with chapter 4. A more thorough introduction to the field can be found in (WEISS, 1999).

2.1 Intelligent Agents

In this section, we will introduce the basic ideas of intelligent agents on a rather high level of abstraction. The main aspect of this introduction is its use of a very general notation scheme for the description of agents in order to be independent of any particular agent school.

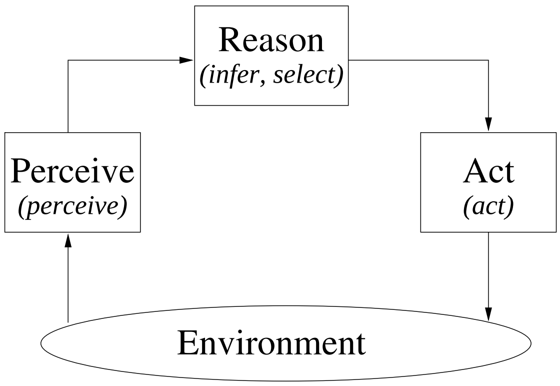


Figure 2.1: Perceive-Reason-Act Cycle

2.1.1 What's an Agent, anyway?

Answering this question, which was firstly asked in (FONER, 1993), is far beyond of the scope of this dissertation as there exist roughly as many definitions of the term as there exist researchers in the field — or perhaps even more. Therefore, we will use a notion of agency that is widely accepted because it covers almost any of the more specific definitions and is therefore not suspicious of being biased towards a particular of those. Basically, an agent is a software system that is situated in an environment and that operates in a continuous *Perceive-Reason-Act (PRA)* cycle as it is depicted in Figure 2.1. Thus, the agent receives some stimulus from the environment and processes this stimulus with its perceptual apparatus. Next the agent starts a reasoning process that combines the newly incorporated information and the agents existing knowledge and goals and this then determines possible actions of the agent. The best of these possible actions is then selected and executed by the agent. The action activation changes the state of the environment which in turn generates new perceptions for the next cycle.

This cycle was already introduced in (GENESERETH AND NILSSON, 1987) and it is also used in (RUSSELL AND NORVIG, 1995). To capture the basic PRA cycle more concisely and formally, we introduce the following notation that is oriented at the formalism used in (GENESERETH AND NILSSON, 1987). One of the basic assumptions in the above description about agents is that they are situated in some environment. We denote this environment by $\mathbf{S}$ as a set of external states without imposing any constraints on the structure of the elements in the set. Then, we describe an agent as a 7-tuple $\langle \mathbf{D}, \mathbf{T}, \mathbf{A}, \mathit{perceive}, \mathit{infer}, \mathit{select}, \mathit{act} \rangle$ where $\mathbf{D}$ is a database that contains the agents acquired knowledge, a set $\mathbf{T}$ of

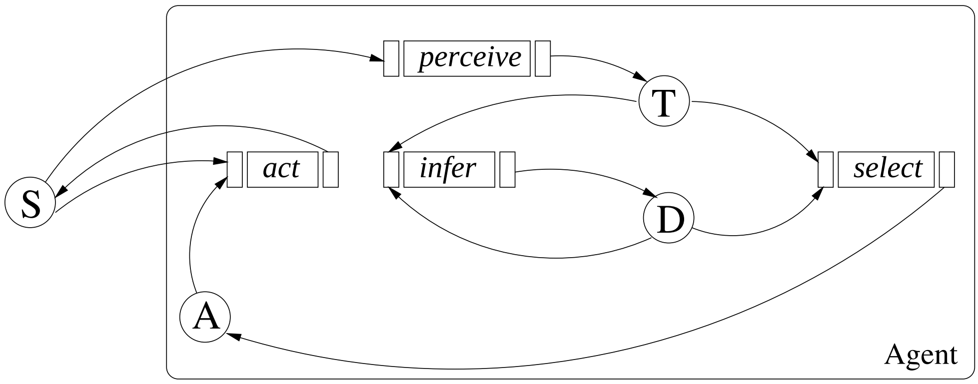


Figure 2.2: A Single Agent

partitions of the environment $\mathbf{S}$ which constitute the possible perceptions of the agent and a set $\mathbf{A}$ of possible actions of the agent. An agent is then defined by the following four functions. The $perceive : \mathbf{S} \rightarrow \mathbf{T}$ function determines how the state of the environment is perceived by the agent, ie. it limits the amount of information that is provided to a partial view on the complete state. The $infer : \mathbf{D} \times \mathbf{T} \rightarrow \mathbf{D}$ function is used by the agent to update its internal knowledge base according to the newly received perceptions. The $select : \mathbf{D} \times \mathbf{T} \rightarrow \mathbf{A}$ function is then used to determine the best action for the current cycle and the $act : \mathbf{A} \times \mathbf{S} \rightarrow \mathbf{S}$ function, finally, changes the state of the environment accordingly. In Figure 2.2, we have depicted an agent that consists of these components and the information flow between them.

To illustrate the ideas that are captured in this rather simple model with an example, consider an automated container terminal where a robot has the task to unload incoming containers from trucks and to store them on shelves in the storage area. Figure 2.3 shows such a facility with several shelves, one robot and a truck that has just delivered some containers that must be unloaded.

Using the abstract description scheme given above, the scenario is modeled as follows. The environment $\mathbf{S}$ of the agent (robot) is a grid world with labeled objects on grid locations, the possible actions $\mathbf{A}$ of the agent are `pick_container`, `drive_to_location` and `drop_container`, the robot's perception $\mathbf{T}$ is the content of the field in front of the robot and the knowledge base $\mathbf{D}$ of the agent, finally contains the destination of each container that is delivered by a truck container.

The Perceive-Reason-Act cycle of the agent is started when the $perceive$ function of the agent determines the presence of newly arrived containers in front

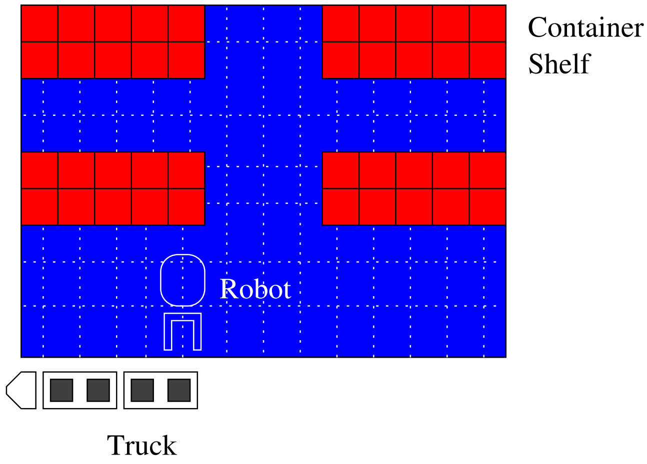


Figure 2.3: A Container Stock Area

of the robot (assuming that the default waiting position of the robot is at the container ramp of the terminal). Then, the $infer$ function decides that the only possible action is to pick up a container which is consequently scheduled for execution by the $select$ function and finally executed by the act function of the robot. As a result of this action, the state of the environment changes (because the robot is now holding a container) and thus the next PRS cycle is started in which the robot will determine the destination of the container and bring it into the storage area.

In this example, the problem solving capabilities that are necessary in the problem domain are directly associated with the agent. This concept, however, has sometimes shown to be too restrictive and it is becoming more and more obvious that an intermediate concept that de-couples the agent from associated problem solving capabilities adds clarity to the modeling process (KENDALL, 1998A). This intermediate concept is called a "role" and it is discussed in the next section.

2.1.2 Roles

What is a role? Unfortunately, this question has no straightforward answer as there exist several definitions of the "role" concept in the agent research community that differ mainly in their focus of what is regarded as important or generic properties.

A very general definition is given in (SUNDERMEYER, 1993) where a role is seen as a primary sociological concept that must be operationalized for the context of agent systems. Thus the question of what is a role cannot be defined in a general fashion but needs the context in which it is to be used. A more specific definition for a role is presented in (WEISS, 1999) where a role is "The functional or social part which an agent, embedded in a multiagent environment, plays in a (joint) process like problem solving, planning or learning. ... ". (WERNER, 1989), on the other hand, limits the concept of a role to purely cognitive states that are defined by the knowledge, the permissions, the responsibilities and the assessment of the agents current situative context.

Probably the best idea to work on a broad definition that is still useful is to start in the field of sociology as suggested in (SUNDERMEYER, 1993). In (BAHRDT, 1994), the major characteristics of a role are given as follows:

- A role is a collection of expectations towards the behavior of the inhibitor of a particular position that allows the members of the society to predict the inhibitors behavior and to plan according to their expectations.
- There exist mutual dependencies between roles, some roles can only exist if other roles do exist as well, for example the role of a "teacher" only makes sense if the corresponding role of (at least one) "pupil" exists as well.
- A member of a society can play several roles even at the same time. This property is called *role multiplicity* and can lead to so-called role conflicts.

A major problem in the field of sociology is the delimination of roles that occur within a society. Not every set of coherent behavior can be regarded as a role, there must exist some special properties that make such a set a role. In developing agent applications, the system designer is faced with a similar problem in identifying coherent sets of behaviors that can be grouped together to form the roles that occur in the problem domain. However, we will postpone this problem to Section 5.4.1 where we will discuss our approach to solving it and instead continue with the more abstract view on agents that we begun earlier.

Formally speaking, we model the concept of a role as an extension of the agents current knowledge, the possible actions and the $perceive$, $infer$, $select$ and act functions. Thus, agents that can play several roles from a set of roles $\mathbf{R}$ are described by the 7-tuple $\langle \mathbf{D} \cup \mathbf{D}_r, \mathbf{T}, \mathbf{A} \cup \mathbf{A}_r, perceive \cup perceive_r, infer \cup infer_r, select \cup select_r, act \cup act_r \rangle$ with $r \in \mathbf{R}$.

To illustrate these ideas, we will add a second role of the robot within the container terminal scenario by extending the original perception function of the agent by the possibility to receive external commands. The (human) area operator can now direct the robot to search for a particular container in the stock and to report the status and position of the container. Thus, we now have two possible roles for the agent, ie. "carrier" or "verifier".

We have already said in the introduction of this chapter that the concepts that will be discussed operate on a rather high level of abstraction. Now, we will become slightly more technical and explain how these theoretical concepts can be mapped into executable machinery.

2.1.3 Agent Architectures

In order to show how the theoretical concepts are actually implemented into computer hard- and software we need an intermediate layer of abstraction that is provided by agent architectures. In (SLOMAN, 1996) an agent architecture is defined "as the portion of a system that provides and manages the primitive resources of an agent". This definition, however, is still to general to be directly applicable and therefore, a two step process is used to bring the conceptual abstractions down to an actual implementation.

The first level of abstraction is given by cognitive models that refine the basic abstractions into more specific concepts. One of the most prominent examples for a cognitive model are BDI architectures (RAO AND GEORGEFF, 1995) that have gained much attention in the agent community in recent years. In the BDI theory, an agent is described by its **B**eliefs that determine the agents current world knowledge, its **D**esires that determine the goals of the agent and finally, the **I**ntentiones that are generated from reasoning about the current beliefs and goals and therewith determine the best possible actions.

But even these more concrete specifications of agents are still difficult to break down into operational concepts. Therefore, a second level of abstraction is necessary that describes how the abstract concepts of the first level are made executable on computer hardware. (WOOLDRIDGE, 1997) suggests three possible means to achieve this goal. The first possibility is functional refinement as it is common in most standard Software Engineering environments, the second one is direct execution of the specifications which implies powerful description languages and runtime environments and the third possibility, finally, is compilation of the abstract specification into executable code. All three of this methods are currently used and none of these has proven to be better then any other.

Because of the huge impact of the decision for a particular agent architecture, Section 5.7.1.3 will provide the reader with a characterization scheme that structures the requirements of the problem domain and that supports the decision for or against a particular architecture. In this section, we will therefore not go into further details but instead discuss the connection between agents, roles

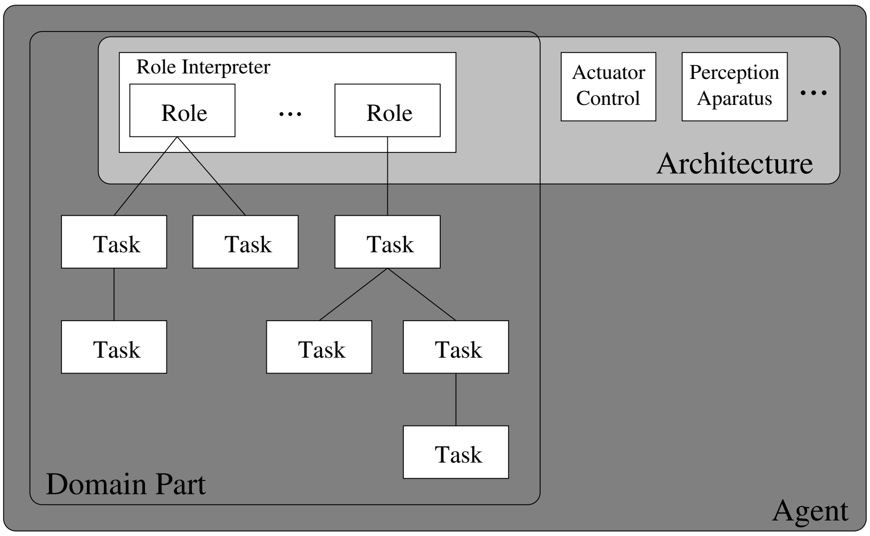


Figure 2.4: Agents and Roles

and architectures in the next section.

2.1.4 Agents, Roles and Architectures

In the previous sections, we have introduced three fundamental aspects of intelligent agents. But how do these concepts relate to each other? Basically, the relation between agents, roles and architectures can be reduced to the following equation.

$$agent = roles + architecture$$

Thus, an "agent" is an abstract concept that is filled with a particular content by defining the possible roles of the agent and by providing a runtime environment that is capable of executing the given role models. In Figure 2.4, the above relation is interpreted graphically. The concept of an *agent* encloses the *architecture* that contains the perception and actuation subsystem as well as the *role interpreter*. The role interpreter links the domain-independent architecture to the domain specific aspects of the different roles by associating each role with a particular task tree.

In the example of the container terminal, the hardware of the robot corresponds to the agent architecture that implements the runtime environment for the possible roles. The roles itself are modeled as task trees, eg. the "carrier" role

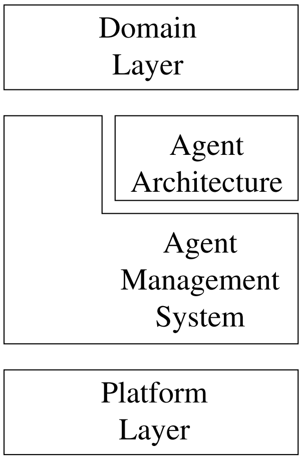


Figure 2.5: A Generic Agent Application Architecture

has the subtasks of checking for incoming containers, determining the destination of each container and then taking each container to the indicated destination.

The above relation between the basic entities in an agent application has some implications on the development of agent (and later multiagent) applications because it determines the basic structure of each application in this class. A possible generic application architecture is depicted in Figure 2.5 where we have shown a conceptual model that consists of three layers. Still, other generic architectures (HORN AND REINKE, 1999) are possible but beyond the scope of this introduction.

The basic layer is of course the *Platform Layer* that hosts the target application. Figure 2.5 shows the rather simple case where the entire applications runs on the same hardware platform. In the case of more complicated applications, it may spread over several platforms and therefore may require an additional layer that provides an abstract interface to the individual platforms. However for the sake of simplicity, we have not included this additional layer.

The *Agent Layer* of the generic application architecture contains two major elements. The *Agent Management System* provides the interface between the agent architecture and the hardware platform and the *Agent Architecture* implements the runtime environment for the domain dependent roles of the agent.

The roles themselves are subject to the *Domain Layer* that covers the domain specific aspects. Note the interface between the agent management system and the domain layer which is necessary whenever the systems functionality is not entirely covered by agents.

In the next section, we will now extend the single-agent case and lift our view one step further to systems of intelligent agents.

2.2 Systems of Agents

In a multiagent system, several intelligent agents exist within the same environment. The term "environment" is hereby used is a very broad sense and covers physical environments for robotic agents as well as runtime environments for software agents, virtual reality environments etc. To express the fact of a shared environment, we describe a system of several agents by a set structure $\{\mathbf{S}, \langle \mathbf{D}, \mathbf{T}, \mathbf{A}, \mathit{perceive}, \mathit{infer}, \mathit{select}, \mathit{act} \rangle_i\}$ where $\mathbf{S}$ denotes the environment just like before and each of the different agents that share the environment has a unique identifier i that distinguishes it from the other agents.

In (FULBRIGHT AND STEPHENS, 1994), other forms of agent coupling have been discussed. However, we will limit our views to this particular form a coupling via the environment because it is predominant in the multiagent research community and none of the other forms presented there has gained general acceptance. The main feature of a system that is comprised of several intelligent entities is that a major part of the systems functionality is not explicitly and globally specified, but that it *emerges from the interaction* between these individual entities. Therefore, interaction is the main aspect of multiagent systems.

2.2.1 Interaction

Coordinated interaction among several autonomous entities is the core concept of multiagent technology. But first of all, what is interaction? To be as general as possible, we define the concept as follows.

DEFINITION 2.2.1 (INTERACTION) *Interaction is the mutual adaption of the behavior of agents while preserving individual constraints.*

This very general definition has some interesting properties that need further clarification. First of all, interaction is not limited to explicit communication or even more specific to the case of message exchange as, of course, the predominant means in the multiagent literature. We define any kind of behavior that is related to other agents as interaction(WEISS, 1999). The example of an ant hill illustrates the basic idea where the single ant does not reflect about the existence of other ants but it still adapts its behavior to the behavior of other ants in a way such that the entire society shows coordinated interaction. The communication between the ants is carried out by several means eg. physical tactile behavior, chemical substances, vision and others.

The second important property of Definition 2.2.1 is the focus on *mutual* adaption, ie. the requirement that the participating agents co-ordinate their behavior. For example, a pedestrian, who jumps out of the way of an approaching car and the driver of the respective car do not have any kind of interaction according to our definition. The pedestrian has unilaterally adapted his behavior in order to avoid a situation that would have yielded a worse payoff to him then to the driver of the car.

The third major focus of the above definition is the aspect of *balancing* between social behavior that is manifested in the mutual adaption and the self-interest of the agent. Neither egoism nor altruism are the best means to achieve globally optimal system states, but a good combination of these two aspects of interaction can yield the best global results (AXELROD, 1984). It is therefore important to equip the agents within a multiagent systems with a mix of self-interest and social consciousness that allows them to value the performance of the entire society over their individual performance.

Especially the second and third of the above properties of Definition 2.2.1 contain the potential for conflicts within the agent society that must be resolved in one way or the other. Possible conflict resolution strategies and generic forms of interaction will be discussed in Sections 5.5.1.1 through 5.5.1.3 and so we will not go into further details here. Instead, we will now turn to structural aspects of an agent society.

2.2.2 The Social Dimension

The social structure of a society determines how the entities within the society relate to each other. Thus, the major questions that occur in conjunction with the social structure of an artificial society are clearly related to sociology and to organizational theory and therefore, we will use some definitions from the field of sociology (BAHRDT, 1994) to explain the basic ideas of agent societies.

DEFINITION 2.2.2 (STRUCTURE) *A structure is a collection of entities that are connected in a non-random manner.*

This definition prescribes two important properties for an agent society that is built upon the definition. Firstly, it requires the agents to be able to perceive existence of other agents, otherwise the term "connected" would not make sense. Secondly, it requires that the agents are arranged towards a particular

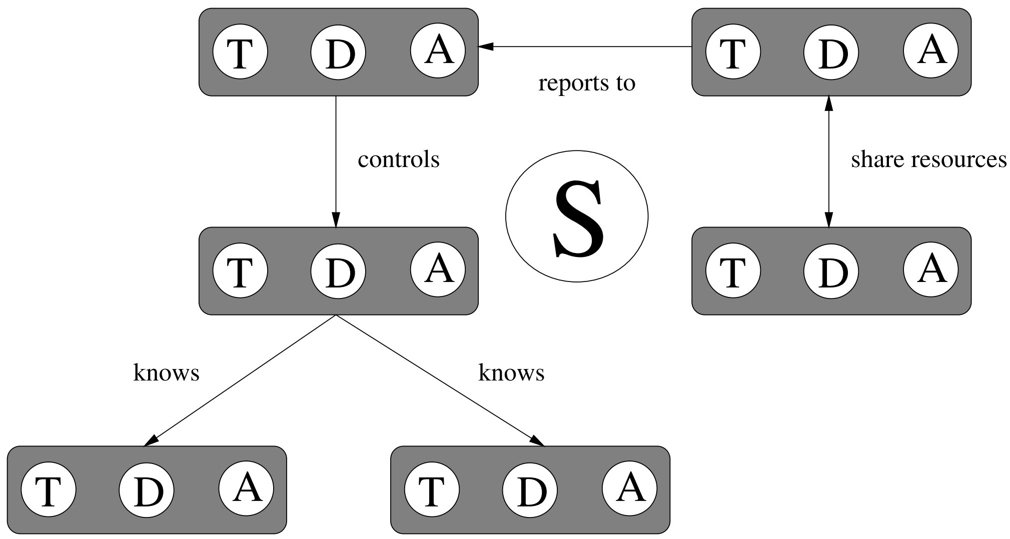


Figure 2.6: Sociogram

intention, ie. any structure must have a purpose. The structural description of the agent society can serve two purposes. A *descriptive* characterization is used to model a society that already exists and which should either be modeled by a multiagent system or that constitutes the organizational context of the system, whereas a *prescriptive* characterization captures the developers intention of how the agent society should look like. Regardless of the purpose of the characterization, however, a sociogram as it is shown in Figure 2.6 can be used to express the structural connections between the agents in an agent society. Each link in the figure has an associated characterization of its meaning that describes the nature of the connection between the entities.

However, having a non-random structure alone does not make an agent society (GERBER, 1997) and thus we must find additional properties that refine the intuitive concept.

DEFINITION 2.2.3 (SOCIETY) *A society is a structured set of agents that agree on a minimal set of acceptable behaviors.*

This is a very general definition of the concept that leaves sufficient freedom for the system designer to model a wide variety of agent societies. The term "acceptable behavior", however, needs some additional clarification. The definition of what is acceptable behavior can be implemented into the agents themselves by the agent designer. The agents then do not have a chance to show non-acceptable behavior. In this case, it is straightforward to achieve acceptable behavior of all agents within the agent society. Unfortunately, this method is only applicable in closed agent societies. In an open agent society such as the Internet, no central definition of acceptable behavior exists. Each agent may have a different view on the topic and the first difficulty is for the agents to agree on a common definition. Furthermore, the agent society must be equipped with punishment mechanisms that can be used against agents that violate the commonly agreed definition. This case is difficult to handle and up to now, no satisfactory solution (especially for punishment mechanisms) has been proposed.

However, having defined the concept of an agent society is still not sufficient to capture the focus of this view. An agent society, just like a structure, does not exist for its own right, instead it must have a purpose. Hence the next definition.

DEFINITION 2.2.4 (SOCIAL SYSTEM) *A* **social system** *is a society that implements a closed functional context with respect to a common goal.*

This definition adds the teleological component to the agent society in that it puts the society into a well defined functional context. Thus the agents within a social system must have a common goal that they pursue as long as they are part of the society.

It is one of the most difficult parts of the development process to find the society structure that is suited best for a particular functional specification because the quality of the solution is usually determined by several, sometimes contradicting, aspects. In Section 5.6, we will outline some of this influential factors and present a micro process model that supports the developer in finding the best society structure for a given problem.

2.3 Summary

We have outlined the basic ideas of intelligent agents and multiagent systems. Starting from a very general formalization of an agent, we have discussed the concept of different roles and introduced agent architectures as the runtime environment for role models. The single-agent case was then lifted to systems with several agents and the fundamental aspects such as interaction and the social dimension of an artificial society have been discussed.

CHAPTER 3

Basic Concepts in Software Engineering

THE SOFTWARE DEVELOPMENT METHOD that is presented in this dissertation is built upon a number of standard Software Engineering concepts and combines them effectively to a single, coherent model. In this chapter, we will summarize the basic concepts and we will briefly introduce the notation used to describe the software design in the case study.

3.1 Aspect-Oriented Programming

Aspect-Oriented Programming (KICZALES ET AL., 1997) is a relatively new idea in Software Engineering that tries to solve a problem for procedural programming languages as well as for object-oriented approaches. The problem that is dealt with are design decisions that effect several, functionally distinct parts of the system at the same time. A prominent example for this phenomenon is error-handling: the error handling strategy of a software system is usually the same throughout the entire system without being explicitly modeled anywhere. Thus it is hard to alter the strategy without major changes to most parts of the system – a process that requires a high effort and that is error-prone. If, on the other hand, the error handling strategy was explicitly modeled, the designer could change it easily and then use tools to re-build the system automatically. It

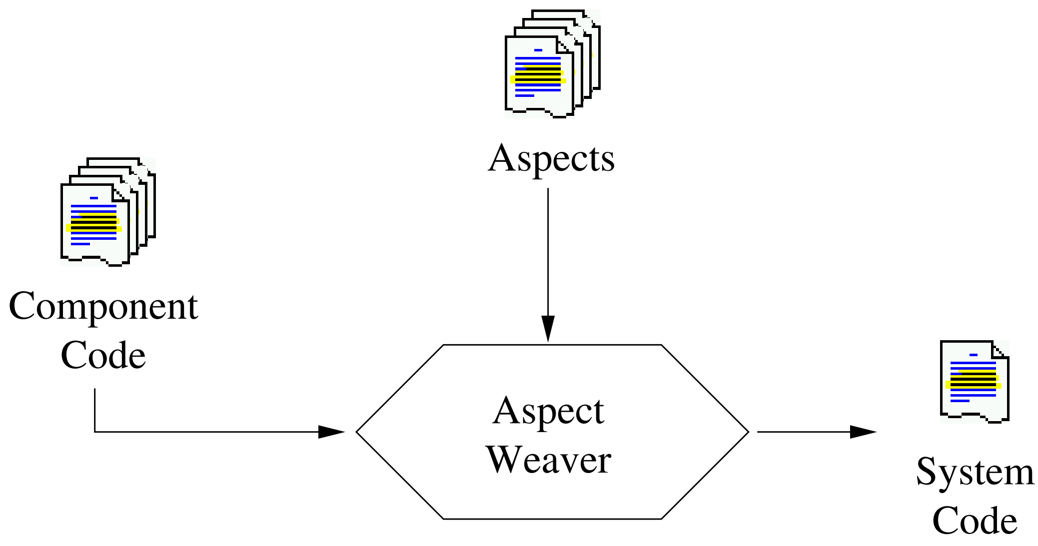


Figure 3.1: Aspect Weaver

is the goal of the aspect-oriented programming research to develop the mechanisms and tools for this process.

Aspect-oriented programming differentiates between two classes of entities that make up the system design. *Components* are the functional units that emerge through the functional decomposition of the target system whereas *aspects* represent the design decisions that cannot be encapsulated into a single functional unit.

The idea of Aspect-Oriented Programming is to express aspects as meta-rules over the set of components and *weave* the system code from the code of the components and the aspects as shown in Figure 3.1.

While this idea seems to be quite natural, it bears a number of problems that have to be solved before aspect-oriented programming can become an accepted and established programming paradigm. The major problem is the mutual dependency of the programming language of the code, the rule language to express the aspects and the aspect weaver itself. In the current state of the development, a rule language and an aspect weaver must be developed for each target language. Furthermore, it is still not clear what expressive power the rule language must have and whether or not all relevant aspects of a particular system can be modeled with the basic approach.

Because of these shortcomings, we will not use the full idea of aspect-oriented programming but rather borrow the basic observation that in every system there exist aspects that *cross-cut* all functional decompositions. Thus,

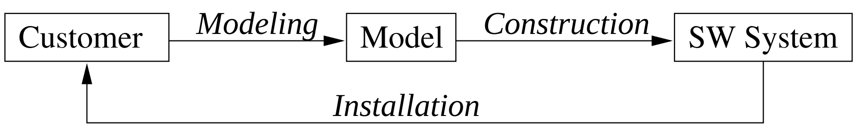


Figure 3.2: Ideal Software Development Process

the functional view on the system and the modeling view are often orthogonal to each other and their connection is seldom obvious.

We will later present a product model for multiagent systems that shares this basic idea with aspect-oriented programming. During the software development process, however, the product model is constructed by the system designer according to a particular explicitly or implicitly defined process model. In order to relate the MASSIVE process model to other existing models, we will discuss some well-known process models in the next section.

3.2 Process Models

In this section, we will sketch some basic process models that have been developed in the Software Engineering community in recent years. The need for prescriptive plans for the development of software systems arose as it became clear that the software development process contains numerous difficulties that cannot be solved in an ad-hoc manner (NAUR AND RANDELL, 1969).

Ideally, software development as shown in figure 3.2 is achieved in three steps:

(1) The designer develops an *abstract model* of the aspects of the real world that should be implemented by the software system.

(2) The system is *constructed* (implemented) according to the model.

(3) The operational software system is *installed* at the customers site.

The products that are generated in the course of this development process are the *model* that describes the designers view on the system and its environment as well as the designers intention of what the system is supposed to do and the *implementation* that realizes the model on a computer platform.

Unfortunately, things are not always that straight because of various difficulties that can occur during the development process. Therefore, the designer needs some pre-defined plan of how to execute the activities that are involved in the software development process and also plans of how to handle difficulties that arise in the course of these activities. Such a pre-defined plan is called a process model as defined below.

DEFINITION 3.2.1 (PROCESS MODEL (JALOTE, 1997))
A Software Engineering **process model** *is a formalization of the software design and implementation activities and of the products that are connected with these activities.*

A large number of different process models have been proposed since the need for these prescriptive plans became apparent. We have chosen some of these models to present them in subsequent sections either because they are fundamental and widely accepted and used in industrial contexts or because they are closely related to our development philosophy.

3.2.1 Waterfall Model

The Waterfall model (ROYCE, 1970) was initially created to describe the various stages that occur during the software development process and it is one of the earliest attempts to formally describe software development. The Waterfall model is a document-oriented top-down approach that models the entire software development process as a sequence of consecutive steps where the output of one step serves as the input of the next step. As depicted in Figure 3.3, the individual steps that are executed during the software development process are the *analysis* of the problem domain, *designing* the software system that solves the problem, *coding* the design into a particular programming language, *testing* the code with respect to the requirements and finally *installing* the system at the user site.

The Waterfall model optimistically assumes that these steps can be executed in the given order, ie. that all documents are complete and no problems occur that would require to trace back to a previous stage.

Due to its simple, sequential nature, the Waterfall model is easy to understand and to use, even by unexperienced programmers. Because of the non-existing interdependencies of the different stages, no complicated coordination among team members is necessary. This might explain why the Waterfall model is still widely in use in industrial projects of almost any size.

However, the Waterfall model has a number of serious drawbacks that limit its applicability. Firstly, it is somewhat unrealistic to assume that all documents are complete and error-free upon the first creation. This assumption is often interpreted as a prescriptive requirement and thus the documents are not allowed

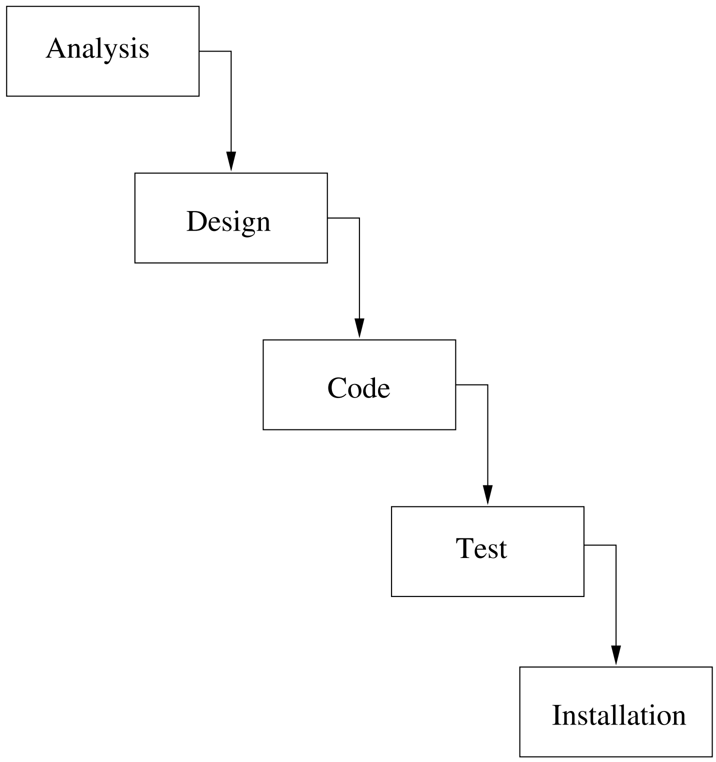


Figure 3.3: Waterfall Model

to change once they are completed. This makes it impossible to correct design decisions that show to be sub-optimal. Secondly, the model does not feature any risk analysis that is helpful in order to identify potential problem areas of the software development process. In a way, this topic is closely related to the optimistic assumption that everything will go all right. The third major problem of the Waterfall model is finally, that the user often tends to see the documentation as more important then the system itself. Thus the focus of the development team shifts away from their initial goal to produce a software system to the goal of producing documents that describe the system.

Due to these limitations and problems of the Waterfall model, it is likely that its application will cease during the years to come and that it will be replaced by more elaborated process models.

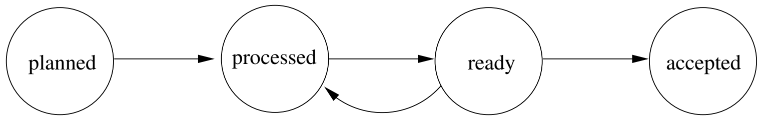


Figure 3.4: Product-States

3.2.2 V-Model

The V-Model (ROMBACH, 1994B), (BALZERT, 1998A) is a direct successor of the Waterfall model and it was designed to fix some of the most obvious shortcomings of its ancestor. As a major enhancement, the V-Model adds explicit quality assurance mechanisms to the Waterfall model: *verification* is concerned with the correctness of individual products with respect to their specifications, and *validation* refers to the correctness of individual products with respect to the intended use of these products. Thus, verification and validation introduce explicit feedback loops if some process stages fail to achieve their goals.

Besides these enhancements, the V-Model provides a more fine-grained view on the process of software development. The basic idea is to model the members of a software development team by explicit *roles* with associated *activities*. These activities can generate or change the *products* that are produced during the process execution.

Each of these *products* has one out of four well-defined states; a product can either be

planned if it does not exist at a particular step,

processed during its construction or refinement,

ready if the construction or refinement is completed or

accepted when it has passed the quality assurance test mentioned above.

The possible transitions between product states are shown in Figure 3.4: When the process enters a state that generates a new product, the state of this product changes from **planned** to **processed** until a first version of the product is assembled and the product state changes to **ready**. If the product succeeds in the following quality assurance test, the state changes to **accepted**. If the quality assurance process fails, the product status is reset to **processed** and a new iteration is executed.

The possible *activities* within the V-Model as shown in Figure 3.5 are *development, verification* or *validation* where either of the activities can

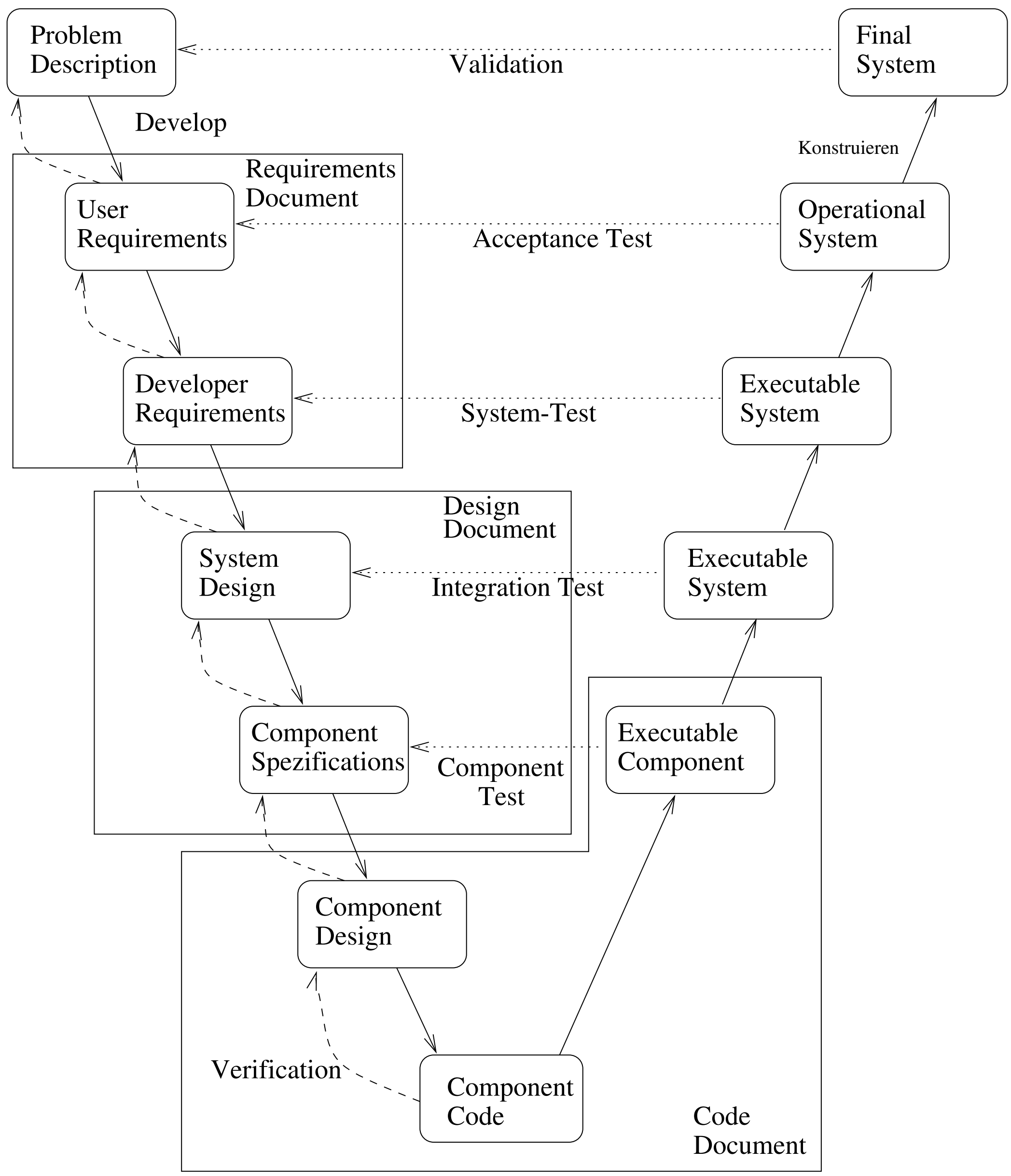


Figure 3.5: V-Model

- generate a product,
- change the content of a product or
- change the state of a product

The generation and content change of a product are usually achieved by the development activities whereas the state-change of a product is associated with either a validation or a verification activity. Each of these activities is exactly documented according to a predefined scheme.

In the V-Model, *roles* are used to describe the required experience, knowledge and abilities for particular activities that are associated with a role inhibitor. The V-Model differentiates between *technical roles* and *management roles*.

The technical roles are the *requirements engineer* who is responsible for the definition of the requirements towards the software system, the *system architect* who designs the software architecture of the target system, the *designer* who is responsible for the design of individual components or entire sub-systems, the *code engineer* who transfers the designers specifications into executable code, the *verification engineer* who is responsible for the correctness of the code wrt. to the functional specifications, the *integration supervisor* who controls the assembly process of components and sub-systems and finally the *validation engineer* who assesses the correctness of the system or parts of the system wrt. to its intended use.

The management roles are the *product manager* who is responsible for a single product, the *project planer* who is responsible for planning the timelines and resource allocations of the project, the *project manager* who controls the project execution and keeps it synchronized with the scheduled activities and resource limitations and the *quality control manager* how is responsible for the maintenance of the quality standards.

The V-model is a very detailed model that is generic and customizable. It works well for large projects and is now a required standard for large industrial projects. It is a mandatory standard for military projects in Germany (and several other countries) and also for many public or government software projects. However, due to its degree of details, it is not suited for small or middle size projects because of the high institutional overhead that proportionally decreases with the project size. Furthermore, the model needs explicit tool support to handle the co-ordination processes between the various roles. Furthermore, the V-Model is not generally method-independent as it depends on a functional and a data-oriented decomposition of the problem to be solved (BALZERT, 1998A).

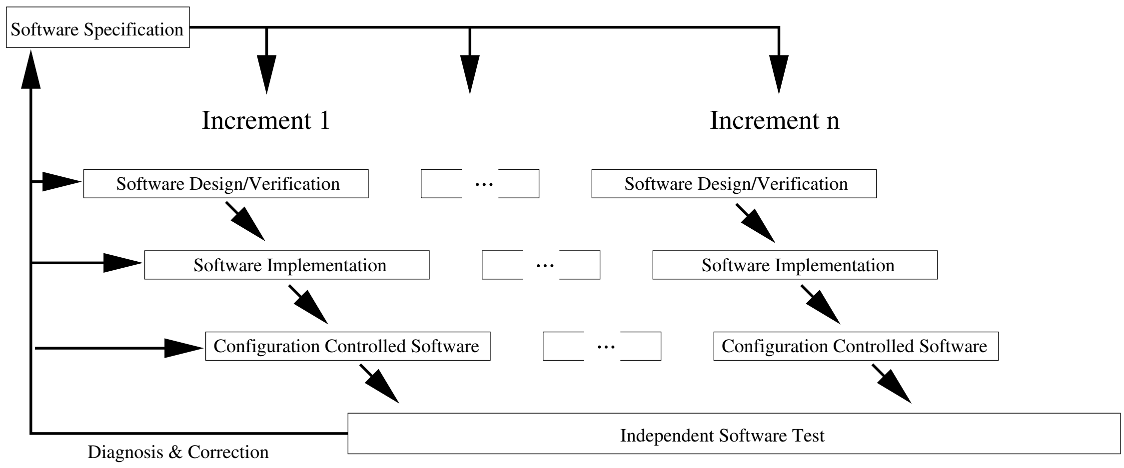


Figure 3.6: Cleanroom Model (DYER, 1992)

3.2.3 Cleanroom

The Cleanroom software development process model (MILLS ET AL., 1987), (DYER, 1992), (POORE AND TRAMMELL, 1996), (LINGER AND TRAMMELL, 1996) was originally designed to support the development of almost error-free software by introducing error prevention instead of error correction in the development process. In most other process models, one or more test-phase(s) are an integral part of the development process itself. The Cleanroom method, on the other hand, takes a different view on the matter of testing. The basic idea is to separate the software development process from the quality management activities. Thus, not the designer or the implementor of a piece of software is responsible for the quality of his or her product, but a completely different authority – the tester. This forces the designer to put more effort into the planning process of the products because errors cannot be "tested out" during the development process. The Cleanroom approach supports this idea by suggesting the use of formal methods of software design as far as possible. The tester, on the other hand, has no connection whatsoever to the design process as he or she is only responsible for maintaining the well defined quality standards through applying statistical testing methods to the software delivered by the designer. This separation of the two major activities in the software development process ensures a higher reliability of the final product and leads to a higher productivity because of the reduced work.

As shown in Figure 3.6 the Cleanroom process model divides the target system into several independent increments that can be separately worked on us-

Process Model 1 Cleanroom

(1) **Requirements specification**

(2) **Development**

  (2.1) Design

  (2.2) Implementation

  (2.3) Configuration management

(3) **(Independent) Testing**

(4) **Quality test**
If quality is sufficient goto (5)
else goto (2)

(5) **Done**

ing an iterative refinement approach. The process model shown in the figure is summarized in Process Model 1.

In the requirements specification phase, the basic increments of the final software system are identified and the requirements towards each of these increments are defined. The specification should use formal method as far as possible in order to allow for test specifications to cover all relevant aspects of the system.

In the development phase, the designers and implementors are not allowed to test their code. Some implementations of the Cleanroom approach even require that the development team is not allowed to *compile* the code produced. In order to prevent the "testing in" of quality, the only allowed techniques for the designers and implementors to check their design or code are code-reading, walk-troughs and formal verification approaches.

The goal of the test phase is to verify that the previously defined quality standards are fulfilled by using functional testing with random input within the specification range. The randomization of the test data should be directed in a way that puts more emphasis on the parts of the system that are more important or more in use then others. It is thus based on the formal specification documents as well as use cases that describe the intended uses of the systems. Bugs that occur within the testing phase are not removed by the testing team. Instead, only feedback for the development team on where the bug occurred is provided.

The advantages of the Cleanroom process model are that it allows for the production of software with very high quality standards and is thus well suited for all kinds of safety-critical applications. The incremental approach that is applied within the development process allows for an efficient resource allocation throughout the entire project life cycle and supports the development of large scale systems.

However, the Cleanroom approach is a very complicated and costly method that requires an appropriate organizational infrastructure as well as experienced staff to work properly. Especially the use of formal methods is currently not wide-spread in industrial software engineering environments. Furthermore, the approach requires a high discipline within the project team to avoid "cheating" between the development and the quality assurance team.

3.2.4 Prototyping

Prototyping (FLOYD, 1983), (LICHTER ET AL., 1994) is a Software Engineering method for the systematic support of the early development of executable software artifacts. Prototyping supports the user-designer communication and is usually used to demonstrate in research and development or in an industrial setting to evaluate requirements and design decisions. The executable system components enable the designer and the user to experiment with requirements and design decisions and to find more appropriate alternatives if some of these decisions do not reflect the intended use or fail to work properly.

Rapid Prototyping can have one out of two intents. Firstly, it can be used to incrementally build a full sized product through the iterative enhancement (BASILI AND TURNER, 1975) of the core system. Secondly, it can also be used to build a scaled down version of the target system in order to demonstrate and investigate crucial points.

As shown in Figure 3.7, two types of Prototyping can be distinguished. Whereas *horizontal* prototypes implement the functionality of a single layer of the final product to its full extent, *vertical* prototypes implement selected functionalities over all layers of the final product.

Prototyping is a valuable tool in order to reduce risks in the software development process because it allows the designer and user to identify critical requirements or design decisions. Therefore, prototyping is often used to support the planning process of a software development project. The early and intensive user interaction and the wide variety of existing tools make Prototyping an important technology besides other software engineering life cycle models.

However, using Prototyping alone is not sufficient for real applications but only in conjunction with some of the other methods describe in this section. First of all, Prototyping often invites the software designer to shift objectives during the software development process. Projects that started experimental version of the final product are silently used as a first version of the target system. However, this is seldom a good idea because of the quick-and-dirty tech-

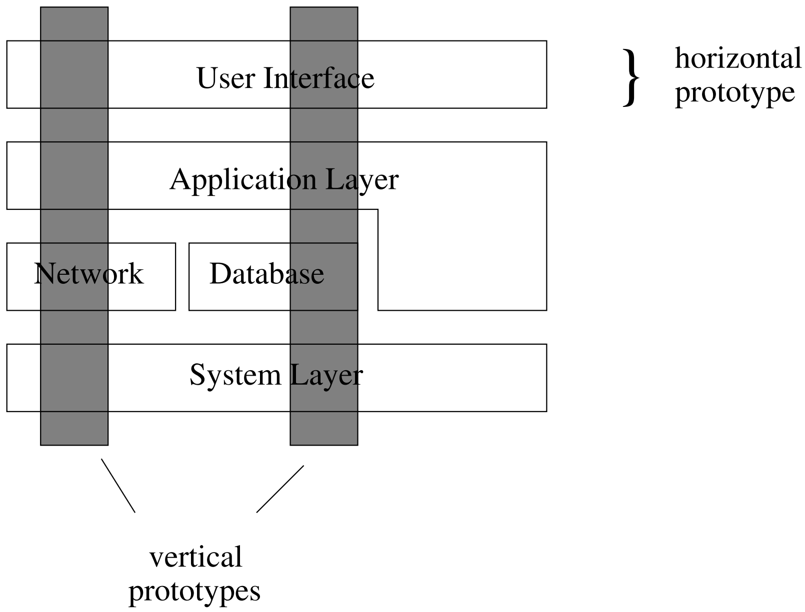


Figure 3.7: Classes of Prototypes (BALZERT, 1998A)

niques that are usually used on such prototypes and because of the poor documentation that is normally available. Thus, the resulting system will contain all these shortcomings throughout its lifetime. Furthermore, Prototyping often suffers from missing user interaction because of either missing interest or because of the wrong institutional settings. Prototyping only works with motivated users that have a strong interest in the functionality of the final product as for example in a research and development environment.

3.2.5 DSDM

The *Dynamic System Dynamic Modeling (DSDM)* method (THE DSDM CONSORTIUM, 1998) is an attempt to define an industrial standard for the development of IT Systems with tight time scales on the basis of *rapid application development (RAD)*. The method features an iterative, product-centered process model that is used to incrementally build the target system, it can be seen as an attempt to formalize the prototyping approach presented in the previous section. The DSDM method is a user-centered approach that relies on the integration of user input throughout the entire software development process.

DSDM is not designed as a general purpose method but instead as a specialized method for business applications where most of the functionality of the target system is visible or accessible through the user interface. Furthermore, the task of the target system should be decomposable into several sub-tasks and the method can only be applied when the classes of designated users are known in advance and when these users are accessible to the development team. If the population of designated users is represented by a small subgroup of users, this smaller group is referred to as the "ambassador user".

The DSDM method is build upon a number of basic principles that must be obeyed to in order to get the method working. First of all, the method depends on *active user involvement* which means that the ambassador users are not regarded as the customers but rather as equal partners in the software development process. Secondly, the software development team (=developers + users) should have *autonomy* in their decisions. It is essential that the development team knows that it is allowed to make any decisions that it regards as essential for the final product. Thirdly, the method is built on the *frequent delivery of (interim) products*. The process by which the product is finally generated can be chosen according to the particular requirements of the product and is not pre-determined by the method. Fourthly, all decision within the development process must be *reversible*, ie. the development team must use some sort of configuration management that allows for backtracking or reconstruction of previous product versions. This is necessary to ensure that potentially risky decisions can be reversed at any time. Fifthly, the method requires *integrated testing and validation* throughout the development process which guarantees that the quality assurance process is built into the development process.

These are the major principles of the DSDM method, there are a number of additional assumptions that the method makes eg. that the requirements of the target system are frozen at a very high level at the beginning of the project or that any deliverable must not be over-engineered in its functionality. These additional issues, however, are only of minor interest and are thus not explained in greater detail.

The life cycle model of a system that is constructed according to the DSDM method consists of five phases as shown in Figure 3.8. The first phase consists of a *feasibility study* that determines the suitability of a RAD approach and describes the technical and managerial conditions of the development process. The second phase is a *business study* that aims at the definition of high level functional and nonfunctional requirements, a system architecture outline and the definition of maintainability objectives. In the *functional model iteration* phase, an incremental prototype is set up that is re-engineered during the *design and build iteration phase*. The goal of this phase is to evaluate the prototype with respect to the needs in the operational context of the customer. The last

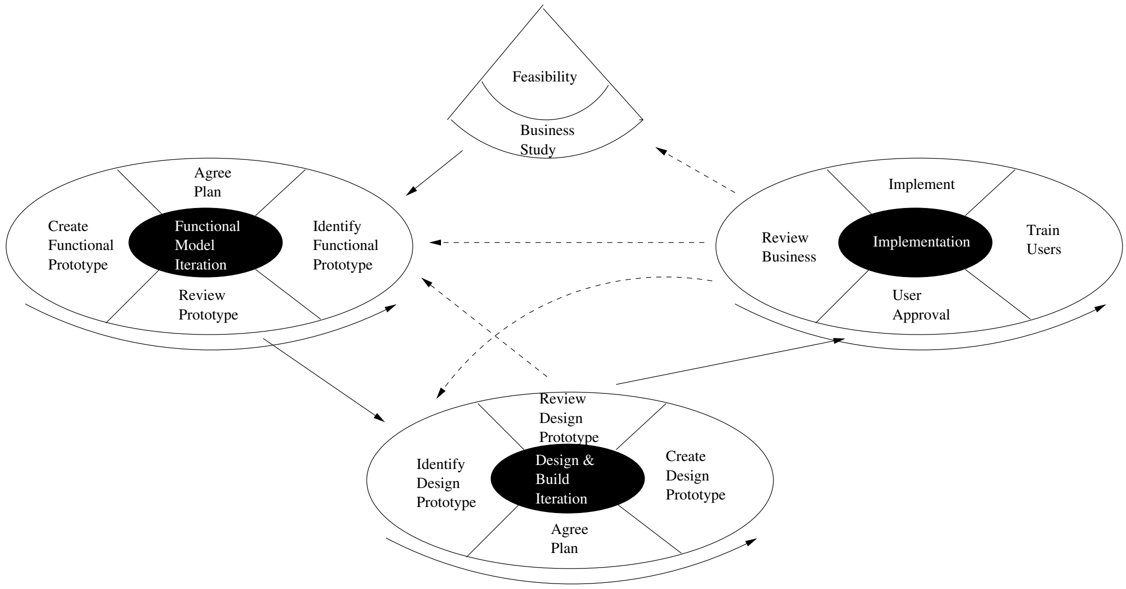


Figure 3.8: DSDM (THE DSDM CONSORTIUM, 1998)

phase, finally, is the *implementation* of the system in the organizational structure of the customer and the user training. During this phase, additional project reviews that aim at learning from the experiences made during the project take place.

The advantages of the DSDM method are that it is far more formal then normal prototyping approaches and that it is independent of particular tools or techniques. Furthermore,the DSDM supports institutional learning which is a matter that is mostly neglected by other approaches. However, the DSDM is only suited for a specific class of applications and due to the heavy dependency on user interactions, it requires a particular institutional framework for the software development process.

3.2.6 Spiral Model

The Spiral Model (BOEHM, 1988) aims at minimizing risks within the software development process by early detection of potential problem areas and by allowing a maximum degree of freedom in the choice of the process for each separate product that occurs within the project. Besides this major goal, the Spiral Model is also designed to minimize the resource usage and to ensure well defined quality standards.

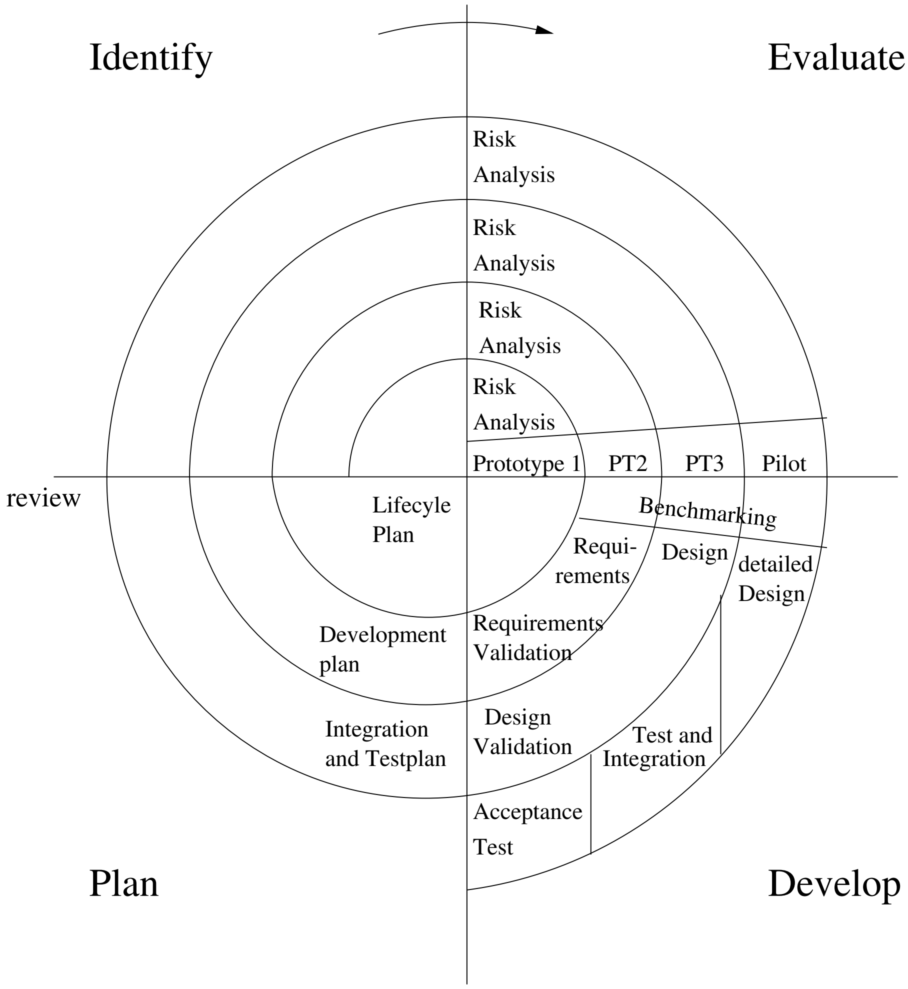


Figure 3.9: Spiral Model (BOEHM, 1988)

The Spiral Model is an iterative model where the inputs of one cycle are the direct output of previous iteration. The process model is oriented at four major activities that are sequentially executed in every cycle. As shown in Figure 3.9, these activities are

Identify This activity aims at the identification of goals, constraints and alternatives for the respective cycle. The goals define what should be done within the cycle and they also set the quality standards that must be maintained for any resulting product. The goals are usually subject to certain constraints that limit the possibilities on how to achieve the goals. However, there may still be several different ways to a goal and thus these alternatives must be outlined be the design team.

Evaluate During this activity, the alternatives that were outlined in the identification step are evaluated with respect to their risks and the implied costs. Risk detection is usually executed by applying prototyping or simulation techniques.

Develop After the most appropriate alternative is selected on the basis of the evaluation of the previous step, the goal products must be constructed. Thus, the first step during the development activity is to choose the most appropriate process model for the goal product. This selection should reflect the constraints and obey the quality standards that were defined at the beginning of the cycle. After the process model is selected, it is executed in order to construct the goal products.

Plan In the last activity of each cycle, the previously executed activities are reviewed and weaknesses that can be identified are reported. Then, the products and resources for the next cycle are planned. An important issue in this step is to commit the entire project team to the plan and to clarify any uncertainties in order to minimize misunderstandings within the project team.

The major advantage of the Spiral Model is that it is capable of an early detection of errors and of ruling out sub-optimal alternatives. Due to the freedom of choice of process models for any product, the approach is very flexible and thus applicable in a wide range of software development projects. Furthermore, the incremental planning process that only considers the next cycle makes it easier to adjust the project plan to changing requirements then in other development process models.

However, this flexibility has its price which is in this case the high management effort that is needed to conduct software development processes according to the Spiral Model. Thus, the approach is not suited for small or middle-size projects.

3.3 Quality Management and Systematic Learning

Quality assurance and quality management are topics that were neglected in the software development process for a long time. Recently, however, the idea of integrating quality measurement in the software development process has shown to pay in terms of faster development cycles and lower maintenance cost. The second important issue that has come up in the software development process is organizational learning. Based on the insight on how dependent an organization is on the knowledge that is informally stored mostly in their human resources, methods were introduced to collect and to enlarge the organizational knowledge in a permanent learning process throughout the entire organization.

In this section, we present two ideas that were developed in the context of software development to ensure high quality software and knowledge preservation over project boundaries.

3.3.1 The Quality Improvement Paradigm

The **Q**uality **I**mprovement **P**aradigm (QIP) (BASILI ET AL., 1994) is a methodological framework for systematic improvement of products and processes within software development projects. The QIP is oriented at **T**otal **Q**uality **M**anagement (TQM) (ISHIKAWA, 1985), a conceptual framework that was developed in Japan and that aims at introducing quality assurance mechanisms into industrial production processes. However, the QIP goes a bit further then TQM as it also defines how to integrate learning feedback into the quality assurance process.

The QIP should only be applied in software development environments that can be unambiguously characterized, ie. that allow for a clear description of all sub-processes and their interdependencies as well as for the clear identification of the products that are involved in the process. Furthermore, the QIP relies on a realistic definition of quality goals. This implies that the desired quality standards must be quantifiable in order to decide whether the process has met the quality goals or not.

The QIP can be viewed as a development process "wrapper" that is put around a particular development process model in order to support systematic learning. Basically, the QIP consists of six steps as shown in Figure 3.10 and Process Model 2.

The QIP features two feedback cycles, one is *project feedback* that is provided during the execution phase (step 4) and the other is corporate feedback that is provided by the models that are constructed on the basis of past experiences. However, no institutional framework is provided by the QIP itself and this makes it sometimes difficult to apply it in industrial projects. To overcome this limitation, the next section will introduce a flexible institutional framework for constant quality improvement and organizational learning.

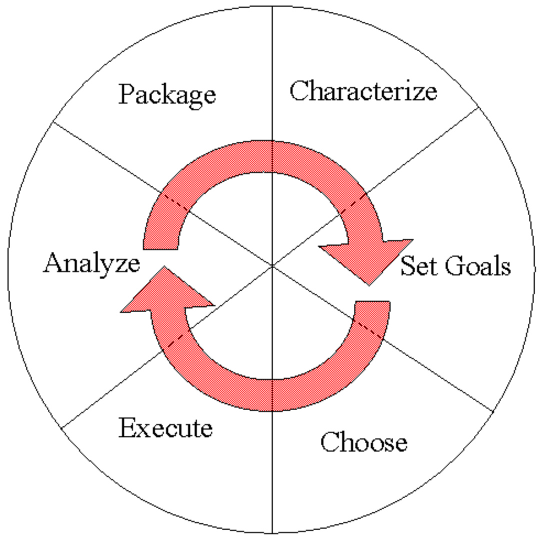


Figure 3.10: The Quality Improvement Paradigm (BASILI ET AL., 1994)

3.3.2 Experience Factory

The *Experience Factory* (BASILI, 1989), (BASILI ET AL., 1994) is a framework that supports systematic learning within an organization and that is already in use in large scale industrial environments (F. ET AL., 1998).

The basic idea underlying the Experience Factory is closely related to case-based reasoning (KOLODNER, 1993) and operates on similarities of software development processes. In case-based reasoning, the specification of a problem to be solved is compared to specifications of problems that were previously solved and if a sufficiently similar problem specification is found, the previous solution is adapted in order to fit the usually slightly different new problem. If the adapted solution is sufficiently different, it is added to the repository for further use.

The process idea of the Experience Factory is almost the same, but instead of just proposing the idea of experience reuse, the Experience Factory also offers an organizational framework for this process. As shown in Figure 3.11, the Experience Factory method differentiates between two organizational substructures: the *Project Organization* is responsible for the project work on the basis of software development process according to the QIP and on the basis of the process models presented in the previous sections. The *Experience Factory*, on the other hand is completely independent of the project organization and is responsible for analyzing and synthesizing all kinds of experiences and to provide

Process Model 2 Quality Improvement Paradigm

(1) **Characterize**
In this step, the goal products, the development environment, available data, used tools, etc. for a particular project are collected and documented.

(2) **Set (quantifiable) goals**
After the characterization is completed, the quality standards must be defined in terms of quantifiable properties of the involved products. The resulting goals should be ordered according to their relevance.

(3) **Choose processes**
The processes that are to be applied within a particular project are chosen in a goal-oriented manner (BASILI, 1993) on the basis of their quality properties and on previous experiences.

(4) **Execute**
The selected processes are executed in order to build the target system. This step is the actual software development process as usually defined in the software engineering literature.

(5) **Analyze**
The resulting project data and determined problems are analyzed in order to make recommendations for future projects with similar characterizations.

(6) **Package**
In this step, either new models are defined or existing models are refined on the basis of the experiences made during the project execution phase.

---

the organization-wide repository of experiences.

Experience packages that are stored in the repository include

Product packages Programs, architectures, designs etc.

Process packages Process models, methods, etc.

Relationship packages Cost and defect models, resource models, etc.

Tool packages Constructive, eg. code generators, configuration management tools, or analytic, eg. profiler, static dependency analyzer

Management packages: Management handbooks, decision support models, etc.

Data Packages Project databases, quality records, etc.

The experience packages can be used to *characterize and understand* new projects , to *evaluate and analyze* existing structures and processes, to *predict and control* the software development processes or to *motivate and improve.*

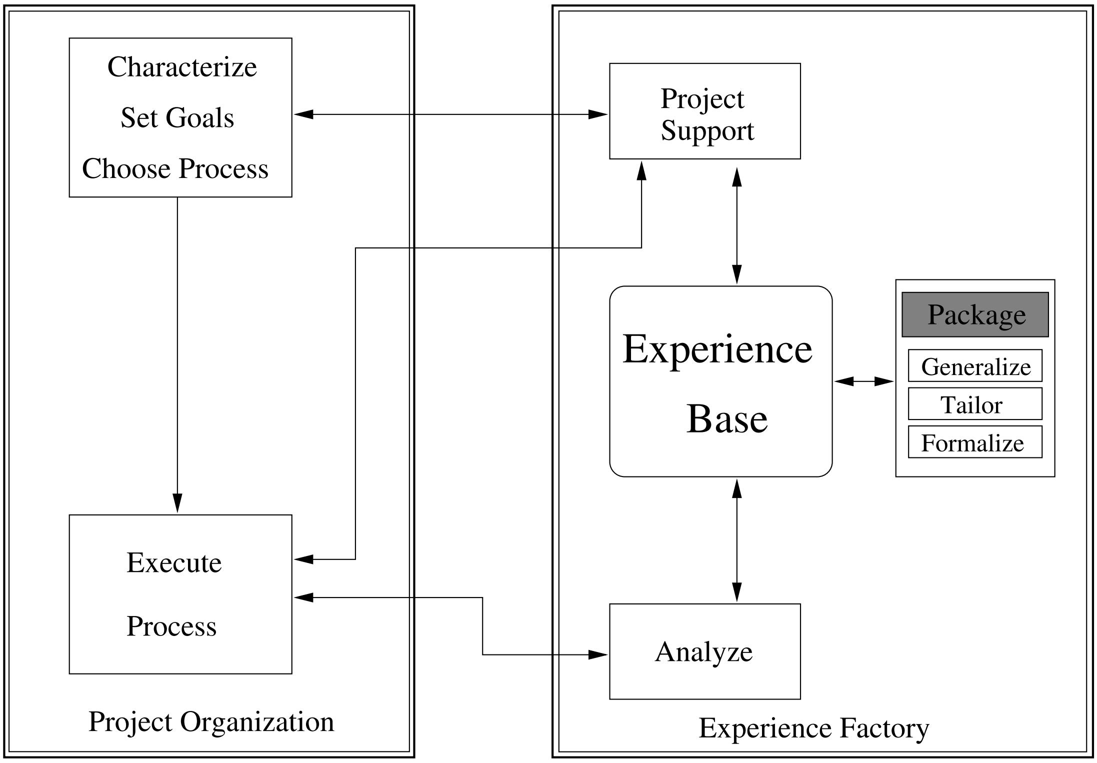


Figure 3.11: The Experience Factory (BASILI ET AL., 1994)

The major advantage of the Experience Factory is that it provides a single interface to the knowledge that exists within an organization and a central authority that collects and updates this knowledge. Furthermore, an Experience Factory can be introduced in parallel to the ongoing business of an organization and does not require complicated synchronization processes nor does it disturb the normal operational business. Finally, the idea of an Experience Factory is highly scalable. It can be used by an individual programmer who will then play several roles in order to store his or her personal experiences for later reuse, or it can be introduced into a large company as strategic department that increases the resource efficiency of the company.

The knowledge structures that will be presented in the MASSIVE method during the next chapters can serve as basic structure for an experience base in order to store the experiences that are made during the development of multiagent systems.

3.4 The Unified Modeling Language

A design tool that has recently gained major attraction is a graphical language to describe the design of software systems. In this dissertation, we will use that language in the case studies to describe parts of the system design.

The **U**nified **M**odeling **L**anguage (UML) (RATIONAL SOFTWARE, 1999A), (BOOCH ET AL., 1999), (RUMBAUGH ET AL., 1999) aims at a global standard for the description of software systems comparable to standardized blueprint languages that exist for electrical, mechanical or civil engineering for several years. The advantage of such a blueprint language for software systems that is built upon a set of symbols and mechanisms together with a well defined semantics for both of them, is that it enables software designer from all over the world to express, exchange and work on their ideas without complicated translation processes that are necessary nowadays. Furthermore, a unified language increases the interoperability among software design tools and allows software developers to become more independent of particular development environments and thus allow them to assemble customized environments out of different tool suites.

The goals of the UML are to provide a ready-to-use expressive visual modeling language that is widely accepted and thus allows the users to exchange design models without loss of information or excessive work to map their models onto each other. Furthermore, the UML should be independent of a particular programming language or development process ie. the UML should be able to support all reasonable programming languages as well as most existing process models. The UML should also provide a formal basis for understanding the modeling language. The formal semantics of the language constructs must, however, not be too complicated so that it can be applied by the average user. UML expresses the operational meaning of most constructs in precise natural language and thus avoids operational definitions that are equivalent to implementational specifications. Extensibility and specialization mechanisms are used in the UML to extend the core concepts by allowing the users to tailor the UML towards their specific needs without the need to alter the core definition or re-implementing tools. The UML also supports higher-level development concepts such as design patterns, components or frameworks to support reuse of models and software and aims at encouraging the growth of the tool market by stimulating interoperability between the products of different vendors on the basis of a commonly agreed format and meaning of modeling constructs. All in all, it is the main goal of the UML to integrate best practices in industry.

The UML is currently undergoing a standardization process that will not be finished for several years. However, the basic functionalities seem to have settled during the past few years and are unlikely to be changed as they are already used in software development environments of all kinds. The building blocks that are currently defined in the UML are described below.

Things Things are first-class abstractions in models that are used to describe *structural*, *behavioral*, *grouping* or *annotational* entities. Structural entities are the static parts (either conceptual or physical) of a software design, eg. classes, interfaces, collaborations, etc.; behavioral things, on the other hand, deal with the dynamic parts (ie. behavior over space and time) and are described in interaction diagrams or state machines; groupings are the organizational parts of the software system (eg. packages) and comments are used to describe, explain or illuminate facts that cannot be expressed in the UML itself.

Relationships The task of relationships is to tie things together by modeling their *dependencies*, ie. the semantic relation between things where the change of one thing may affect the semantics of the other thing; their *association*, ie. structural relationships that describe a connection between things (eg. aggregation); *generalizations* that express the substitutability of a thing (the parent) for another thing (the child) and *realization*, ie. a semantic relationship wherein one classifier specifies a contract with another classifier that guarantees to carry it out.

Diagrams Diagrams are used to group interesting collections of things. The diagram types that are defined within the UML are

- *Class diagrams* for classes, interfaces and collaborations and their relationships,
- *Object diagrams* for objects and their relationships,
- *Use case diagrams* for actors and their relationships,
- *Sequence diagrams* for messages between objects with a focus on the time-ordering,
- *Collaboration diagrams* for messages between objects with a focus on the structural aspects,
- *Statechart diagrams* for a dynamic, event-oriented view of the system,
- *Activity diagrams* for emphasizing the control flow among objects,
- *Component diagrams* for the static implementation view and
- *Deployment diagrams* for the configuration of the run-time processing nodes and the components that live on them.

These concepts will not be explained in this work and we refer the reader to (BOOCH ET AL., 1999) for a full introduction to the UML. We will use the UML in this work to present some of the ideas and the resulting implementation of the case study at a high level of abstraction. These rather simple diagrams should be understandable without an intensive knowledge of the UML.

3.5 Summary

In this chapter, we have outlined major Software Engineering concepts that form the basis of the MASSIVE method. We have presented Aspect-Oriented Programming, a design technique that conceptualizes the idea of collecting several cross-cutting aspects of the software design in a single abstraction and process models that are formalizations of the software design and implementation activities. Finally, we have discussed some aspects of quality management and organizational learning and given a brief overview of the UML.

In the next chapter we will demonstrate how these concepts are adapted to the specific case of multiagent systems and how they are assembled into a uniform framework.

CHAPTER 4

The Conceptual Framework of MASSIVE

THE IDEAS AND CONCEPTS of the previous chapter constitute the basis of general Software Engineering methods which will now be refined for the particular case of multiagent system design. To this end, we will now show how the basic concepts and ideas of the MASSIVE[1] method are derived from these generic base concepts and how they are combined into a coherent method that can be used to develop multiagent applications.

4.1 Features

The fundamental building blocks in the MASSIVE method are *features*. Features can represent conceptual entities such as design decisions, constraints or test cases, or they can represent physical entities such as components, devices or code fragments. We will refer to the features of the first class as *model* features and to the features of the second class as *implementation* features. Furthermore, features can be classified as *task specific* features that describe properties or entities that are specific to the task at hand or as *task independent* features that describe general properties which do not pertain just to a particular application but to an entire class of applications.

[1] MultiAgent SystemS Iterative View Engineering

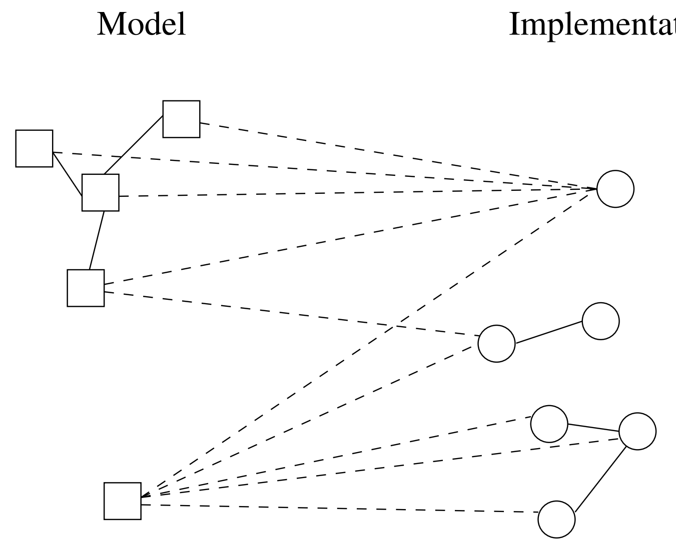


Figure 4.1: Connections between Model and Implementation

Consider, for example, a compiler application. The basic structure of any compiler is almost the same, the main differences come up in conjunction with particular source and target languages. Thus, a constraint such as “the compile time must be less the 30 seconds” is an example for a task independent model feature because the compile time is – except for very exotic languages – independent of the syntax of the source language. Consequently, the syntactical analyzer is a task independent implementation feature as it can be parameterized with the syntax table of the source language. The syntax specification of the source language, on the other hand, is a task specific model feature as it must be developed specifically for each programming language. The code generator for the target language, finally, is a task specific implementation feature because it is usually impossible to have an efficient code generator that operates on transformation tables and thus the code generator must be written specifically for a particular target language.

The MASSIVE method sees the model and the implementation that realizes the model as a single collection of features. The features of the model and the features of the implementation are linked in a flexible way that allows features of one group to be mapped onto features of the other group. This mapping is not unique as shown in the *Feature map* in Figure 4.1 where a single feature of the model is linked to several features of the implementation and vice versa. A collection of model or implementation features is called a *feature group*, ie. a feature group is limited to a particular class of features. The links that are

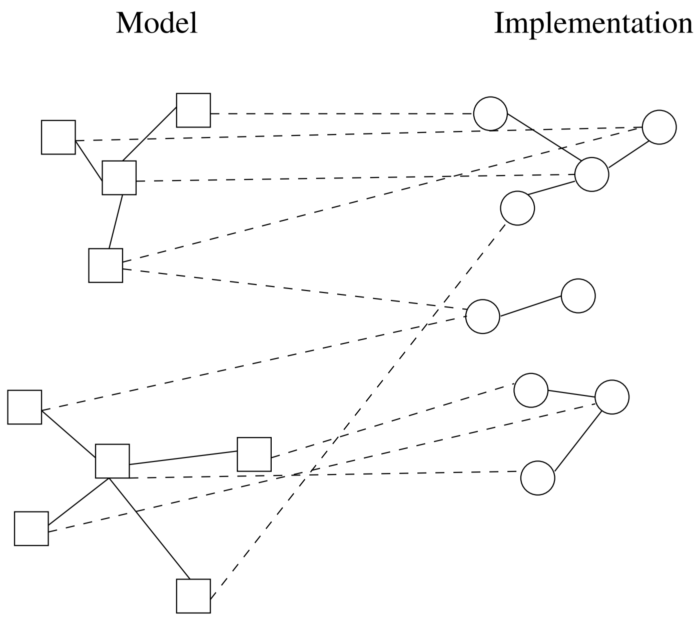


Figure 4.2: Feature refinement

shown in Figure 4.1 represent an association among the features that can have several meanings that usually depend on the type of the involved features. For example, a link from a model feature that represents a particular algorithm to an implementation feature means, that the implementation feature represents the code that implements the algorithm. If, on the other hand, the implementation feature represents a particular hardware device, the link to the model feature usually means that the model feature depends on particular properties of the hardware device and that a change in the hardware causes a change in the model.

Features are recursive structures that can be refined or aggregated. The transition from Figure 4.1 to Figure 4.2 illustrates the idea of feature refinement. The single feature in the lower left corner of Figure 4.1 is refined into five model features in Figure 4.2 and the single implementation feature in the upper right corner of Figure 4.1 is refined into four implementation features in Figure 4.2. The inverse process where several features are combined into a single feature is called feature aggregation.

The MASSIVE method allows to define, alter, refine or aggregate features up to the desired level of abstraction and it is the task of the system designer to identify the relevant features of the target system and to find a structure for the software systems that implements these features.

4.2 Views

In this section, we will discuss the fundamental abstractions used in the MASSIVE method to analyze and design the target system from different points-of-view. We will first introduce the basic idea in a very general sense and then refine the generic ideas for the case of multiagent applications.

4.2.1 What and Why?

Features are a valuable tool to describe and to model the basic properties of a software system, but they are too general a concept for particular application areas. Therefore, we need a terminology that is more closely related to the particular need of multiagent systems design.

The terminological framework that we propose in the MASSIVE method is based on so-called *views* where each view represents a set of conceptually linked features. A view is thus a projection of the model onto a particular subject. In analogy to the fact that there are usually several abstractions from the same reality (PERNICI, 1990), we will later introduce a number of views that describes the target application from different perspectives. Views are conceptually related to aspects introduced in Section 3.1 and usually cross-cut functional boundaries. A collection of views that achieves a logical decomposition of the target system is called a *view system*.

Note the difference to the concept of feature groups that were introduced in the Section 4.1. Firstly, feature groups are limited to a particular class of features (either model features or implementation features) and secondly, feature groups only capture the refinement process that is performed during system development.

Views, on the other hand, can spawn over several feature groups and are thus a more general concept of grouping things together under a common semantic concept. (BALZERT, 1998B), for example, differentiates between four different views that a developer can have on the target system. As shown in Figure 4.3, these four views are the *Data view*, the *Function view*, the *Dynamics view* and the *User Interface view*. Although it is claimed there that these views can be applied to any application, we will later argue why we think that a more specific system of views that is oriented at the basic requirements of the problem class is more adequate.

Ideally, the target system can be decomposed in several independent units with well defined interfaces. These interfaces are modeled as explicit connections between views by the use of so-called *shared features*. Shared Features belong to two or more views at the same time and represent the interface between

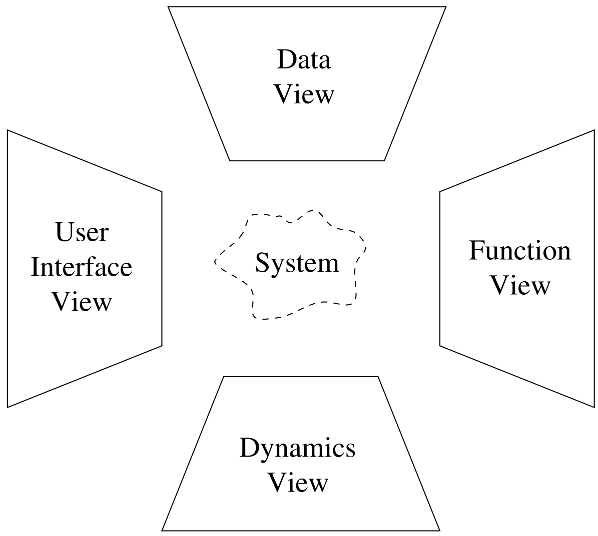


Figure 4.3: A Generic View System (BALZERT, 1998B)

their parent views.

The basic ideas are illustrated in Figure 4.4. View A (indicated by the dark grey squares and circles) and view B (indicated by light gray) have a shared feature in the upper right corner that indicates that these two views jointly determine or depend on a particular entity of the implementation.

It is important to note, that the property of a feature to be shared between several views is not inherited by features that refine the shared feature. As illustrated in Figure 4.5 for a feature that is shared between two views and that is refined by two new feature, the new features can (a) belong to one of the views that share the parent feature, (b) belong to different views (c) can be shared between the views or (d) can be shared with a third view. The non-inheritance of the shared property of a feature ensures the proper encapsulation of the abstractions within the feature and is necessary to maintain the separation-of-concern among the views as far as possible.

4.2.2 A View System for Multiagent Systems

Views are a general concept to arrange features with respect to a logical decomposition of the application class and we will now instantiate the general framework for the application class of multiagent systems. Therefore, we need to define the goals that should be achieved with the newly developed view system.

First of all, the view system should help the designer to analyze the system

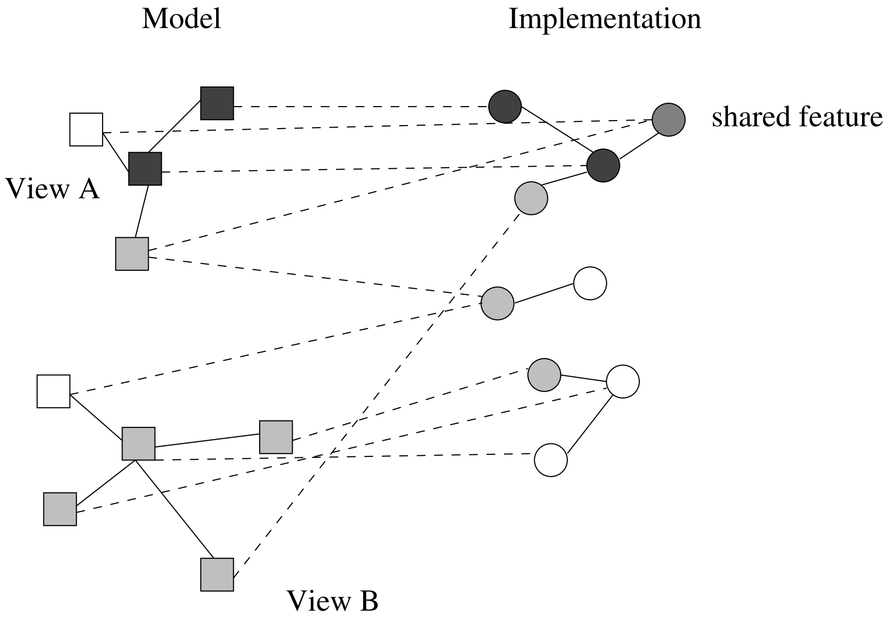


Figure 4.4: Features and Views

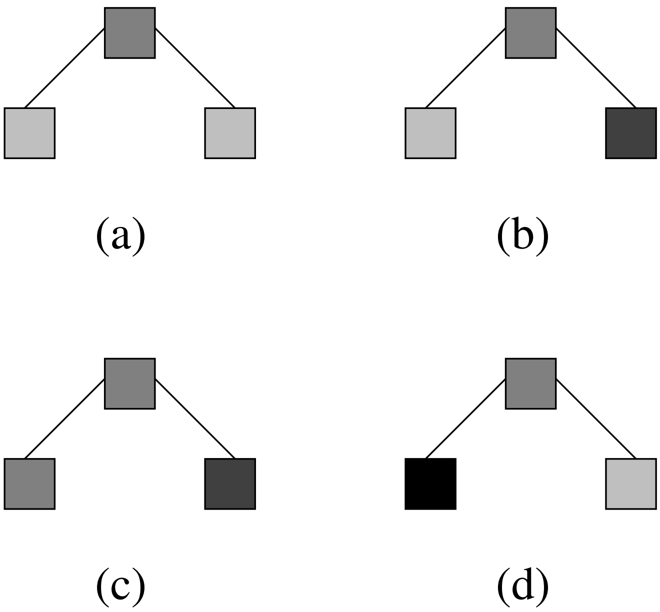


Figure 4.5: Shared Features and Feature Refinement

with respect to four fundamental questions:

- What should be done?
- Where should it be done?
- Who should do it?
- How should it be done?

These four questions cover – at a very high level – all relevant aspects of the target system. Clearly, the focus of these questions is too broad and thus we need a more fine-grained conceptual framework that *separates* the features of the target system as far as possible, *covers* all relevant aspects of the target system, is *flexible*, has a reasonable *size* (ie. is not too detailed neither is it too coarse-grained) and has a *crisp naming scheme* that provides intuitive access for the developers. The basic requirements will be explained in the subsequent paragraphs.

Separation Separation in this context means, that the view system should support "separation of concern", ie. changes in one part of the model should affect as few other parts of model as possible. In the feature oriented approach we have presented in the previous section, this property can be expressed by using as few shared features as possible. The view model presented in the section is a generic model, and thus it can only try to group generic features together into individual views.

Generally speaking, the view model should provide a high cohesion among the features that are grouped together in the view and achieve a low coupling with other views.

Coverage The view system must be able to cover all aspects of the target system. For example, the view system should reflect the fact that the target system is not developed or used in isolation but rather in a well-defined development and operational context. Thus the view system must allow to specify and model not only the properties of the target system but also the properties of the environment of the target system.

Furthermore, the view system must allow the designer to specify functional as well as nonfunctional properties of the target system.

Flexibility The view system should allow the designer to model a broad range of multiagent systems ranging from multi-robot applications to systems of directly communicating software agents. Furthermore, the view system should not be limited to a particular technology, eg. a specific agent architecture or a particular agent framework.

Size The size of the view system is a crucial factor with respect to acceptability in the development community. If the system is too fine-grained it is likely to be too complicated to be used. If, on the other hand, the model is too coarse-grained it will not be used because it provides no added value to the designer in comparison to an unstructured approach.

Naming The names that are used within the view system should be suggestive, ie. they should cover the semantic attributes of the objects referred to as far as possible. The they should also be distinguishable in a way that the designer is able to clearly associate a particular group of attributes with each view.

The view system that we will present now has matured over several years during our work on the design and implementation of multiagent systems. The system consist of seven views that are briefly described in the following paragraphs, a full description of each view together with the application of the views to a case study are presented in the next chapter.

Task view In the Task view, the functional aspects of the target system are analyzed and a task hierarchy is generated that is then used to determine the basic problem solving capabilities of the entities in the final system. Furthermore, the nonfunctional requirements of the target system are defined and quantified as far as possible. Note that this view does not assume that a multiagent approach is used for the final system and therefore provides a rather high-level analysis of the problem.

In the case of a compiler application, for example, the basic functional requirement is that the system translates a program specified in a high-level language to a particular assembly language. The quality of the resulting code or the maximal tolerable time for the compilation are nonfunctional requirements and the basic problem solving capabilities are for example lexical analysis or code generation.

Environment view In this view, the environment of the target system is analyzed from the developers perspective and from the systems perspective. These two perspectives usually differ as the developer has global knowledge whereas the system has only local knowledge. In the RoboCup domain (NODA, 1995), for example, the developer has access to the complete

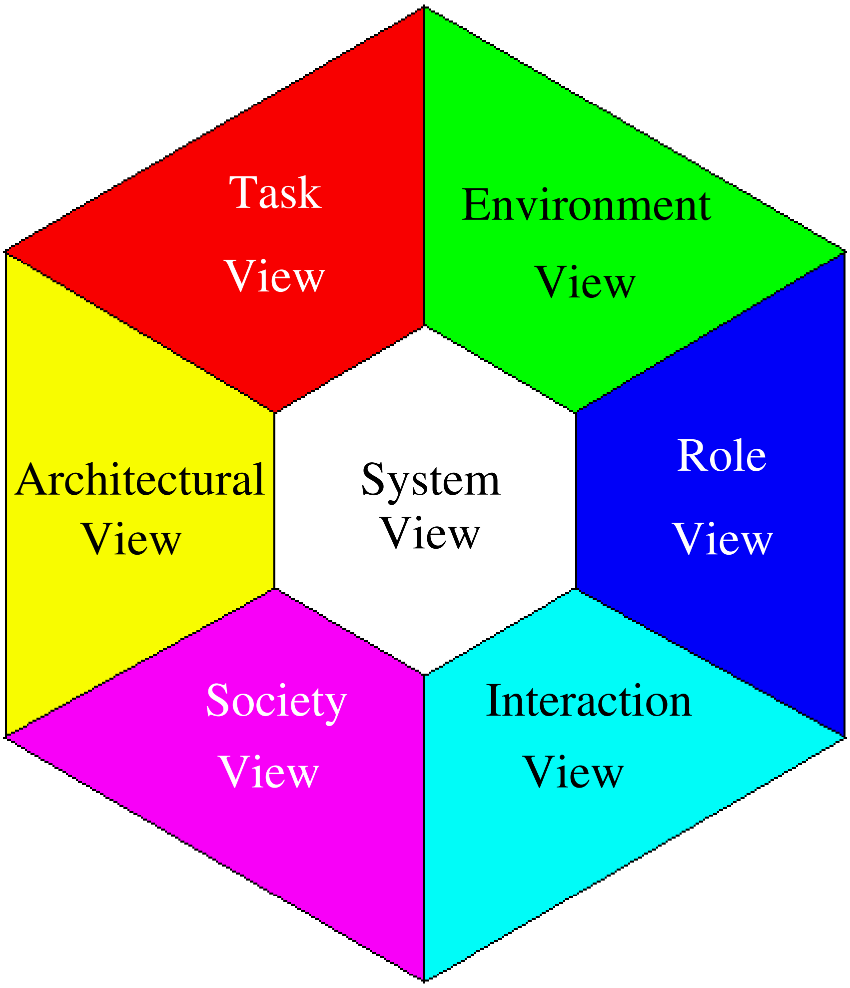


Figure 4.6: MASSIVE Views

state of the system and its environment and this state is completely deterministic from this point-of-view. From the perspective of the individual agent within the system, on the other hand, only parts of the environment are accessible and the state transitions appear to be nondeterministic because of the ongoing activities that cannot be perceived by the agent.

Role view This view determines the functional aggregation of the basic problem solving capabilities according to the physical constraints of the target system. A role is an abstraction that links the domain dependent part of the application to the agent technology that solves the problem under consideration. In our view, an agent consists of one or more role descriptions and an architecture that is capable of executing these role models which makes it important to aggregate the basic capabilities according to physical constraints.

In a robotics application for a storage area, for example, we may find robots that are capable of carrying containers from one area to another and others that are capable of stacking containers onto each other. Therefore, the roles of "carrier" and "stacker" cannot be assigned to a single agent because of the physical constraints of the robots unless a third sort of robot exists that can execute both basic problem solving capabilities.

Interaction view Interaction is a fundamental concept for a system that consists of multiple independent entities that coordinate themselves in order to achieve their individual as well as their joint goals. In this view, interaction within the target system is seen as a generalized form of conflict resolution that is not limited to a particular form such as communication. Instead, several generic forms of interaction exist that can be instantiated in a wide variety of contexts. The developer is encouraged to analyze the target problem with respect to the applicability of these generic forms before designing new forms.

The most popular example for interaction is of course a communication protocol, simply because communication protocols have been studied for quite some time. However, multiagent systems that simulate physical environments or real physical multiagent systems such as robots or machines have many other possibilities of interaction besides communication and these forms of interaction must be allowed for in a general purpose method as well.

Society view A society is a structured collection of entities that pursue a common goal. The goal of this view is to classify the society that either preexists within the organizational context of the system or that is desirable from the point-of-view of the system developer. According to this classification and to well defined quality measures for the performance of the target society that depend on application specific aspects, a society model is developed that is consistent with the roles within the society and that achieves the defined goals.

To illustrate how the quality measure affects the desirable society structure, consider, for example, Internet trading. In order to achieve the best trade, the number of participants in the trading process should be rather high in order to increase the chance of finding a profitable trade. On the other hand, a high number of participants also increases the computational and communicational overhead and thus a clustering of trading agents would increase the communicational and computational efficiency of the system. The final structure of the agent society (flat or clustered) thus depends on the quality measures (quality of the solution vs. efficiency).

Architectural view The architectural view is a projection of the target system onto the fundamental structural attributes with respect to the system design. The major aspects that are dealt with in this view are the system architecture as a whole and – due to the size and complexity of this particular aspect – the agent architecture. The system architecture is described according to various aspects and includes things such as agent management or database integration. The required agent architecture is characterized according to the requirements of the problem to be solved and it is strongly recommended that the system developer should at first try to select one of the numerous existing architectures before trying to develop a new architecture from scratch.

An important aspect that has to be dealt with in this view is to find the appropriate segregation between agents and objects. Just because agents provide a means for structuring a problem does not mean that they are necessarily the best means to do so (COLLINS AND NDUMU, 1998). Sometimes, it is better to implement particular abstractions as ordinary objects and thereby increase the system performance by avoiding the inevitable overhead associated with turning an object into an agent.

System view This view, finally, deals with systems aspects that affect several of the other views or even the system as a whole. The System view, for example, handles the user interface that controls the interaction between the system and the user(s) whose the task specific aspects are usually the input specification and the output presentation whereas task independent aspects deal with the visualization of the system activities in order to enable the user to follow the ongoing computations and interactions. Other aspects that are described in this view are the system-wide error-handling strategy, performance engineering and the system deployment once it has been developed.

The complete view system is shown in Figure 4.6, where the Task view models *what* the system should do, the Environment view describes *where* the system should perform its tasks, the Role view defines *who* should do what and the

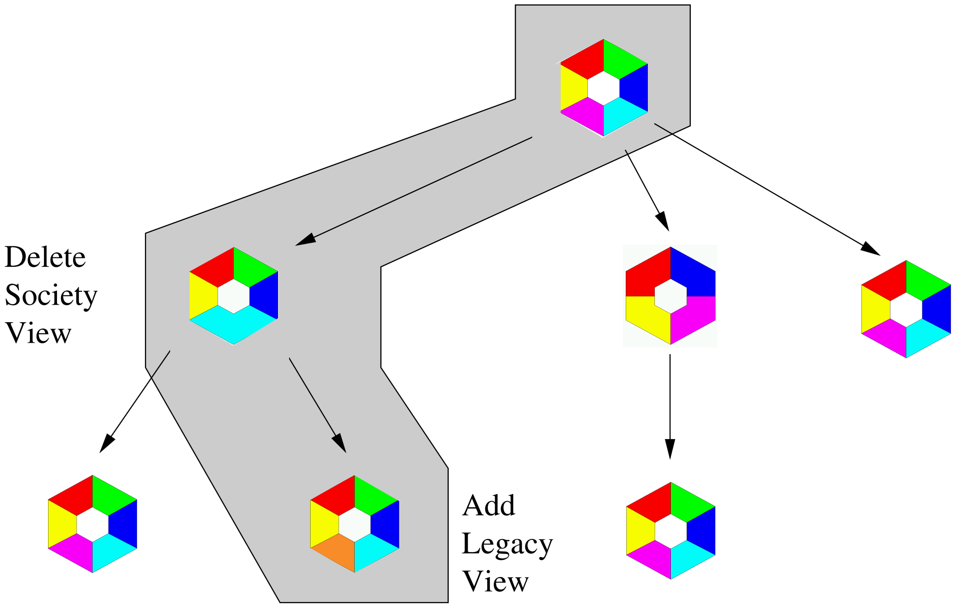


Figure 4.7: The View System Tree

Interaction, Society and Architectural views, finally, define *how* the tasks should be executed.

Furthermore, the view system shown in Figure 4.6 is not stable over time. Rather, it is intended as the root of a tree of view systems as shown in Figure 4.7. The tree emerges from the generic base model by adding view systems that have additional views or view systems that have one or more views deleted.

Consider, for example, a multiagent system that has only a small number of agents. In such system, the Society view may be omitted because of the limited size of the agent society. A followup project has similar characteristics as the original project (ie. few agents) but it may furthermore need on some sort of legacy software. In this case, the Society view is omitted as well, but also a new view, the Legacy view which deals with the integration of legacy systems in the multiagent environment, is added to the view system. The development of the original view system to the new view system is illustrated in Figure 4.7 by the shaded area.

The tree of view systems reflects the evolutionary idea of the Experience Factory presented in Section 3.3.2: the initial view system is altered in order to make it fit the particular needs of a specific project and then the adapted view system is filed away in case that a later project will have the same characteristics as the project for which the view system was originally adapted. The goal of this

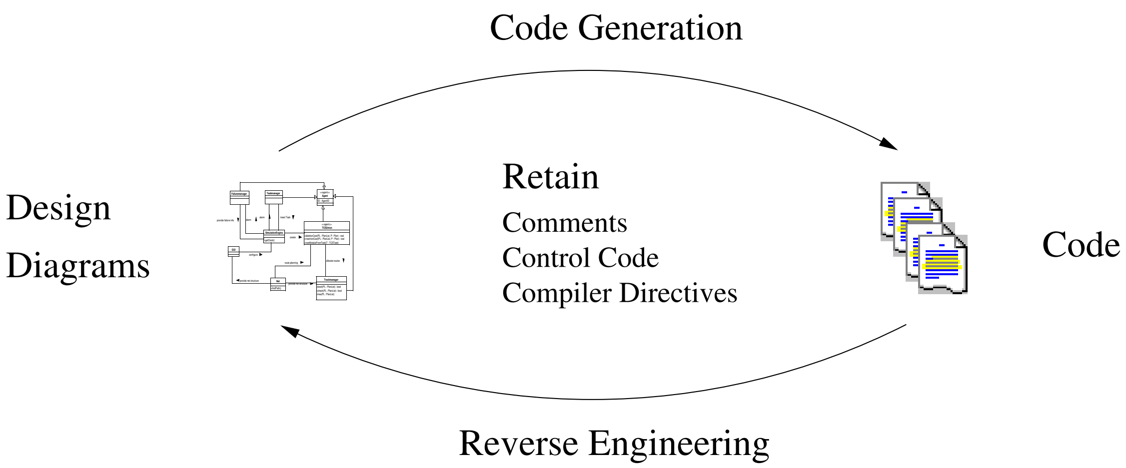


Figure 4.8: Round-trip Engineering

constant learning process is to acquire a collection of view systems that cover a large variety of multiagent application domains.

After having presented the innermost element of the development model, we will now present the process model that uses the view system discussed in this section.

4.3 Iterative View Engineering

Iterative view engineering is a product centered software engineering process model that combines Round-trip Engineering (BALZERT, 1998A) and Iterative Enhancement (BASILI AND TURNER, 1975).

Note (jl): *IVE is closely related to DSDM but is more strict in the definition of the refinement and feedback/re-engineering steps*

Round-trip engineering as shown in Figure 4.8 is itself a combined method that consists of a generative step where the goal is to construct a software system from its specification and an analytical step where the starting point is existing code and the goal is to derive the specification. The first step is called *Forward Engineering* and the second step is called *Reverse Engineering*.

The other software engineering technique that we have incorporated into our process model is called Iterative Enhancement. Iterative Enhancement is a process model that is similar to the Waterfall Model of Section 3.2.1 except for

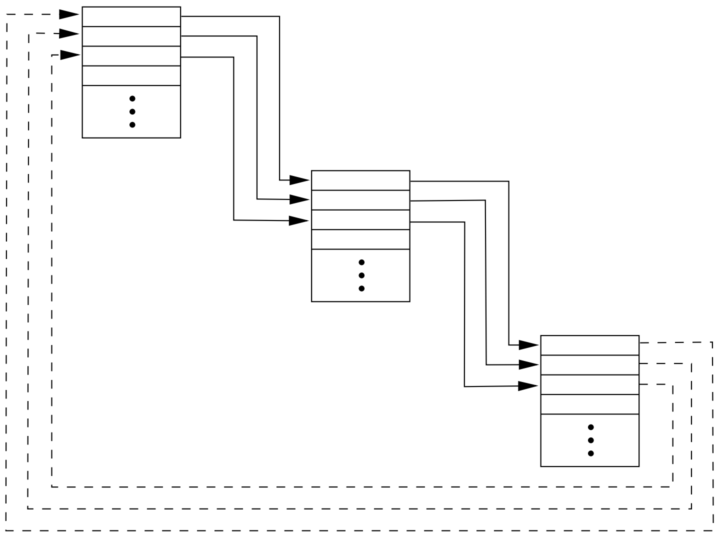


Figure 4.9: Iterative Enhancement

the fact that the Waterfall Model needs a complete and stable requirements definition while the Iterative Enhancement model operates in several cycles on a partitioned and incomplete system model. The incomplete model is iteratively enhanced (hence the name of the method) in each cycle as shown in Figure 4.9 for a three step process. The major advantage of this approach is that it takes into account that any specification is initially incomplete and usually undergoes frequent changes because of upcoming new information, external factors or simply because of human errors (PARNAS AND CLEMENTS, 1986). Although the idea of Iterative Enhancement is pretty old in terms of computer science research, it is still developed further and used in software development methods such as described in (BOOCH, 1996).

Figure 4.9 assumes that all artifacts of the model are constructed with the same process model which is not always optimal. Therefore, we propose an extension that sees the Iterative Enhancement as a *macro* process that encapsulates several *micro* processes that are used to construct individual parts of the model. This idea is illustrated in Figure 4.10 which shows a product model that consists of three parts where each part is iteratively constructed with potentially different process models. Part 1 of the model is constructed according to process model A which consists of 5 different activities, part 2 of the model is constructed according to process model B with 6 activities and finally part 3 that is constructed according to process model C with 5 activities. The vertical boxes represent snapshots of the entire model at times t_1 and t_2, where the parts

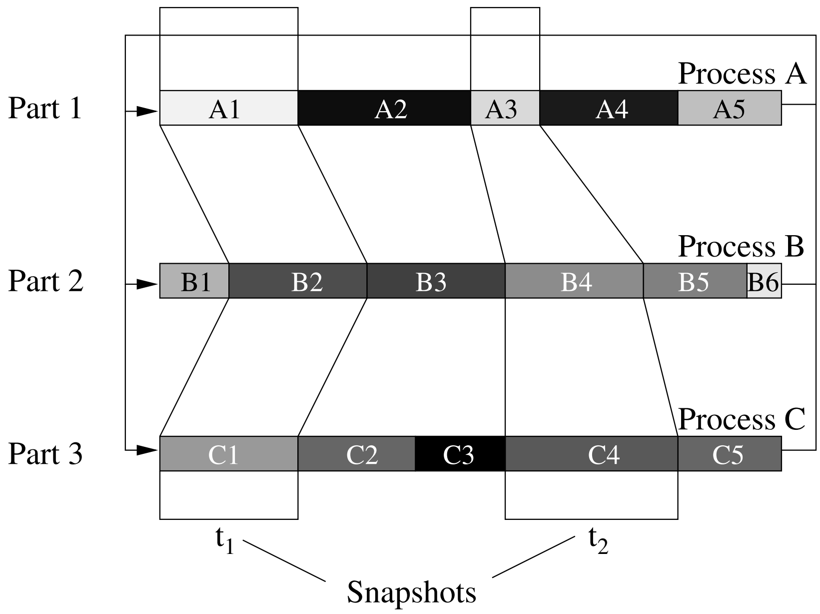


Figure 4.10: Micro Processes

of the model are in different stages of the processes that are used to construct the respective artifacts.

Combining these ideas into a joint model results in the Iterative View Engineering model shown in Figure 4.11. The Round-trip steps are oriented along the horizontal axis of the figure and the Iterative Enhancement steps are depicted on either side of the figure.

The Iterative View Engineering process model is the following: Initially, the model and the implementation are empty. In a first cycle of Iterative Enhancement (①), the designer specifies the first version of the model probably by using different micro models for each view. Note that it is not compulsory for the designer to work through the views in a predefined order, but experience suggests that a cyclic order is quite useful and note also, that the enhancement step operates on the complete set of views. In the following step (②), parts of the the initial model are implemented. If an error occurs during the construction phase, the model has to be refined (③) until it can be implemented. Next, the initial implementation is tested. During the test phase, it often turns out that the model specification was incomplete (even for the parts that were generated) or wrong. In this case the implementation is changed or enhanced (④) in order to meet the intentions of the initial model. After the test and enhancement phase of the implementation is complete, the results must be integrated into the model during

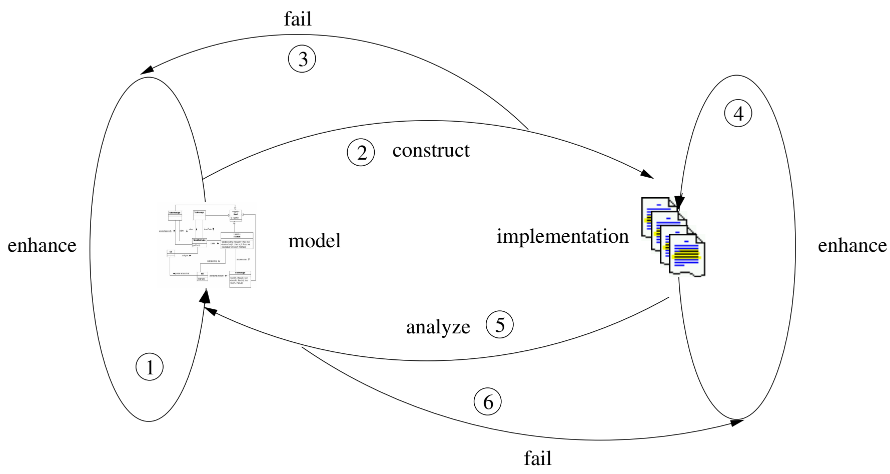


Figure 4.11: Iterative View engineering

an analysis step (⑤). If the implementation cannot be re-engineered (eg. because the changes of the code are incompatible with some basic requirements of the model), the implementation must be changed (⑥) in order to comply to the modeling language. After this step, the next cycle is executed until the entire system is fully implemented.

The advantages of this life-cycle model are that it can deal with an incomplete problem specification because of the iterative enhancement approach. The model is also useful for early risk detection because parts of the model are incrementally implemented and can direct the project managers view to critical regions.

4.4 Putting It All Together

Now that we have seen the individual concepts and entities that are part of the MASSIVE method, it is time to assemble these fragments into a coherent picture.

Figure 4.12 shows the overall structure of the MASSIVE method. The product model that was described in Section 4.2 is integrated into the process model from Section 4.3 which is itself embedded into the framework of the Experience Factory outlined in Section 3.3.2.

The resulting overall development model is given in UML notation in Figure 4.13 and a textual description is provided in Process model 3.

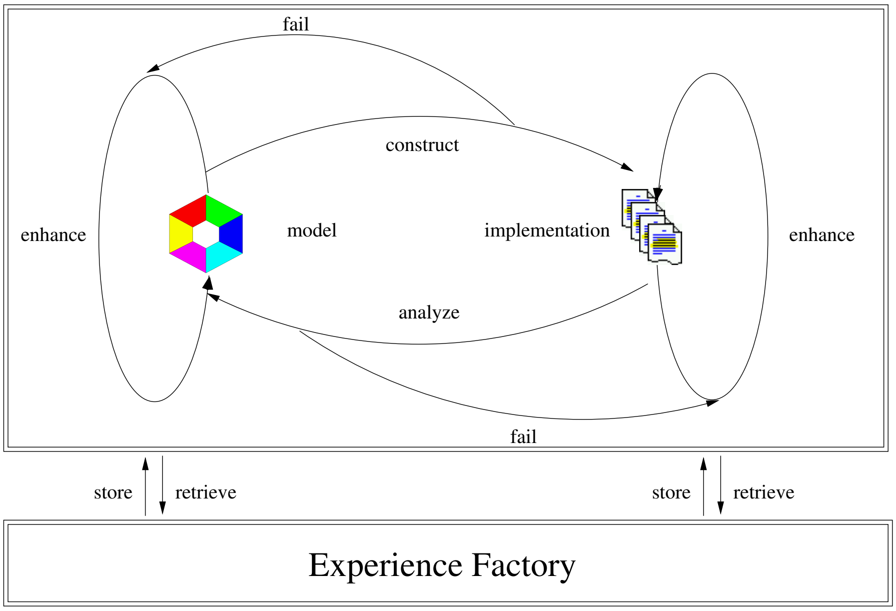


Figure 4.12: The MASSIVE Method Model

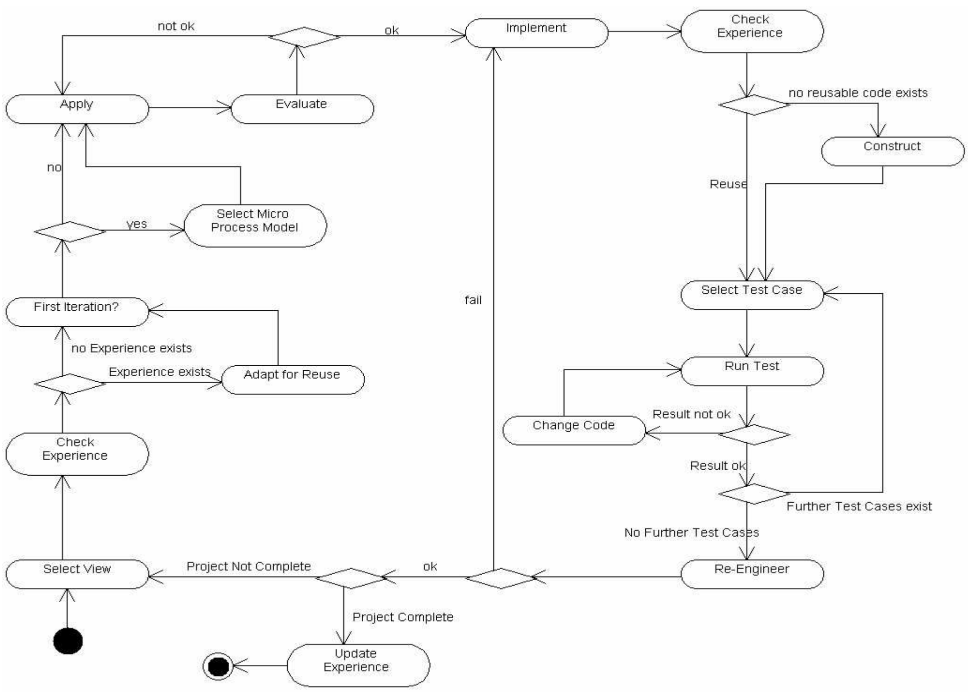


Figure 4.13: The MASSIVE Method as UML Diagram

Process Model 3 MASSIVE

(1) **Select View** In the first step of the Iterative View Engineering process, the developer selects the view that will be refined to from the view system that describes the current state of the development process.

(2) **[Select micro process model]** The micro process model assists the developer in the refinement of the previously selected view. Usually, the micro process for a particular view is selected upon the first iteration of the macro process and remains fixed until the end of the development process. It is, however, possible to change the micro process that is used for a view when it turns out, that the initial decision was not adequate.

(3) **Check Experience Base** Before a view is initially developed or refined, the Experience base is checked for packages that contain knowledge about artifacts with similar specifications. If one or more of such packages are found, the results that are stored in these packages is adapted to the specific needs of the new problem and reused to construct a solution for the problem in question.

(4) **Apply** In this step, the previously selected micro process model is applied to the selected view in order to generate the first version of the view or to refine an existing version either from a previous iteration or from the Experience Base.

(5) **Evaluate** The current version of the view is evaluated according to well-defined quality measures. If the quality is not sufficient, the micro process is re-applied to the view until the quality standards are met.

(6) **Implement** If the view has reached a sufficient degree of maturity, it is transformed into executable code. In this step, the Experience Base is used for the second time in order to find knowledge packages on the code level that can be reused for the implementation. If no matching code fragments are found, the implementation must be started from scratch.

(7) **Test** The goal of the test phase is to ensure that the code that has been constructed in the previous step complies to the specifications that are captured in the model.

   (7.1) **Select test case** The test cases are specified in features that are associated with each view. The Test Engineer selects one of these features for each round of testing.

   (7.2) **Run test** Running a test means to set up the system environment and system state according to the test case specification and then executing the code. The results of the test execution are the analyzed to discover errors in the implementation.

   (7.3) **Adapt** If the results obtained in the previous step differ from the expected results as specified in the test cases, the code must be changed until the results from rerunning the test provide the desired result. If the results match the specifications, the test phase is complete.

(8) **Re-engineer** The test phase of the system can result in changes of the code that are not captured in the model. Thus, this step aims at identifying these differences and adapting the model in such a way, that it adequately describes the current code of the system. This step is necessary to ensure consistency of the model and the code.

(9) **Iterate** If not all specifications are implemented and have passed the quality checks, the entire process is repeated from the first step.

(10) **Update Experience Base** After the completion of the entire process, the Experience Base is updated according to the new experiences that were made during the development of the new system.

The core of the MASSIVE method is the view system that was outlined in Section 4.2 and that will be discussed in detail in Chapter 5. The view system is the conceptual basis for a wide range of product models that are developed and refined throughout the software projects that are carried out according to the suggested process model. The main idea of the process model is an Iterative View Engineering approach that is itself based on Iterative Enhancement and Round-trip Engineering. However, as we have said before, this macro process contains itself several small micro processes that are used for individual activities within the design process for a particular view.

In Section 5.4.1, for example, we will discuss a micro process model for the role modeling activity within the design of the Role view. Such a nested process model was also suggested in (BOOCH, 1996) but there only a single micro process model is embedded into the macro process.

The macro model as well as the micro models of the MASSIVE method are in no way fixed for the entire lifetime of the project model, they are also subject to changes and refinements during the course of time. In order to preserve these and to make them accessible to others, the process model and the product model are both embedded into a larger organizational structure called the Experience Factory. The Experience Factory provides the formal framework for a permanent learning process that takes place over project boundaries and that eventually models the multiagent experience of an organization in terms of specific product and process models for various domains.

4.5 Summary

In this chapter, we have presented the basic ideas of the MASSIVE development method such as *features* that represent conceptual or physical entities, eg. design decisions or system components. Features are conceptually linked in *views* that constitute a projection of the complete model onto particular aspects. *Iterative View Engineering* was then developed as the process model that is used to construct the product model that is built upon these views in the course of the software development process. This overall framework is finally part of the *Experience Factory* which supports systematic learning over project boundaries.

In the next chapter, we will focus on the view-oriented product model that was already outlined in this chapter. Each view will be discussed in detail and the basic ideas will be illustrated using the TCS/MAS system as a case study.

CHAPTER 5

MASSIVE **Views**

THE PRODUCT MODEL OF MASSIVE is the core concept of the entire method. It allows the system designer to break the target system down into several views that concentrate on particular aspects of the system and abstract away from others. In each of the following sections, we will at first discuss the general nature and the intended scope of a view and also discuss a number of features and design patterns that belong to a view. However, the features and patterns that are discussed are by no means an exhaustive collection, they rather represent the current state of our knowledge and experience in designing multiagent applications. The potential user of the MASSIVE method is encouraged to add new aspects that are necessary or to remove aspects that are not important in a particular context according to the basic idea of the Experience Factory. In the second part of each section, the general considerations are applied to the case study to demonstrate how the theoretical concepts are used in a practical situation.

Before we present the views that constitute the view system of the MASSIVE method, however, we should clarify how these views relate to multiagent specific questions. Almost any of the subsequent views covers some agent or multiagent specific aspects of the target application but the degree varies greatly over the set of views. In Figure 5.1, we have sorted the views of the MASSIVE Product model according to their relation to multiagent technology.

The most general and thus most technology independent view is the Task view because it models the system from a purely functional point-of-view. The

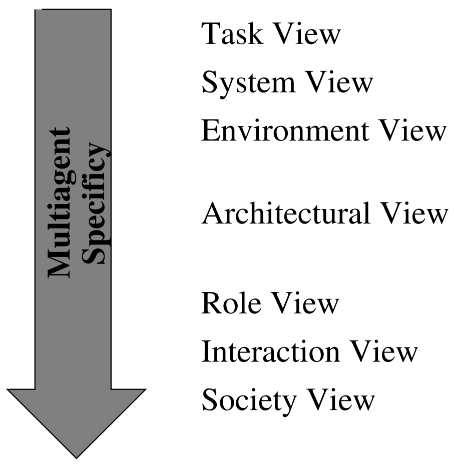


Figure 5.1: Views and Multiagent Systems

System view and the Environment view also deal with very general properties of the target system and contains only few multiagent specific features. In the middle of the ordering, we find the Architectural view that can be interpreted as the link or the interface between the more technology independent views and the more specific views. The first view in this class is the Role view that explicitly models the agents and their properties, followed by the Interaction view, responsible for the ongoing inter-agent processes. The most multiagent specific view, finally, is the Society view that explicitly deals with collections of agents.

Although it seems to be a natural idea to construct the target system by proceeding from the general to the specific views, this approach is not the best for building the target system. This is simply due to the fact that some decisions in a more general view are refinements of aspects of a more multiagent specific view. For example, the roles and interactions in the target system constitute design decisions that have major influence on the architecture of the final system.

Furthermore, some of the views are more associated to the analysis phase of the software development process while others belong more to the design phase. In Table 5.1, we have marked the respective views of the MASSIVE product model according to the phase of the traditional software development process model that they are more closely related too. The relations that are indicated in the table, however, are not exclusive as a particular view may contain analysis and design aspects at the same time. The ordering of the following sections is thus the suggested order in which the different views of the MASSIVE

	Task View	Environment View	Role View	Interaction View	Social View	Architectural View	System View
Analysis	X	X					X
Design			X	X	X	X	X

Table 5.1: Views and the Traditional SE Process

method should be handled. Still, according to the principle of selecting the best strategy for a given problem, the suggested ordering can be changed if necessary.

Before we discuss the views that constitute the multiagent system for the problem under consideration, however, we will provide a brief introduction to the problem that is used in the case study to illustrate the basic properties of each view.

5.1 A Brief Introduction to Train Coupling- and Sharing (TCS)

Efficient transportation — be it of persons or goods — is a key issue in todays industrial world (CARROUÉ, 1997). Because of the immense amount of transportation tasks, it is necessary to use the available resources most effectively. Thus, computer aided — or entirely controlled — scheduling systems are key technologies not only for telematics.

Conventional wagon load traffic as it is performed today is shown in Figure 5.2: a company that wants to ship something via rail to its customers delivers the freight to the local freight center (usually a railroad station) where it is stored until enough freight from other companies has arrived to justify a train to the regional freight center. At the regional freight center, wagons from other local freight centers that have the same direction are assembled and sent to the next interregional freight center where another re-assembling process takes place. The decomposition of the trains is achieved in reverse order.

This approach has some serious drawbacks. First of all, the wagons of individual customers must wait at the local freight centers until enough freight is delivered to make a train to the next local freight center profitable. Secondly, the re-assemblance of trains in regional and interregional freight centers is a very time consuming process that introduced additional delays in the producer-to-customer route.

An alternative approach (KRACKE ET AL., 1995), (FABEL, 1996), (VOGES AND

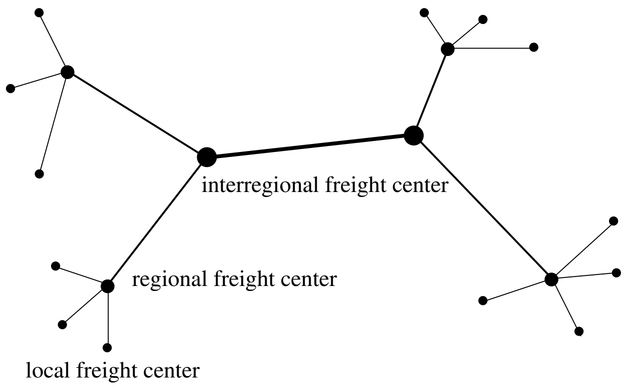


Figure 5.2: Hierarchical freight haulage

MIERAU, 1997) to the classical freight transport process uses small railroad *transportation modules* (eg. (WINDHOFF AG, 1996)) instead of conventional trains. Whereas a normal train is made up of one locomotive and several freight wagons, a transportation module consists of two power units on either side of the module and up to three permanently coupled intermediate vehicles with a fixed number of loading spaces. Thus, a transportation module is a single unit of limited size. When a company wants to deliver some freight to a customer, it orders a transportation module at a local freight center and loads its goods onto this module and the module itself is then responsible to find its way through the railroad network.

A transportation task is served by a transportation module, we assume that each task can be served by a single module, i.e. there is no need to hook two or more modules together to serve a single task. Vice versa, we assume also that a module cannot serve more than one task at a time. All tasks occurring in the system are transportation requests in a railroad network; a network is graph consisting of several nodes connected via so-called *location routes.* In the current version of the system, we use an abstracted map of the German railroad network with approximately 250 nodes and 350 links.

Whenever the system receives a transportation task, it assigns the task to a free module which in turn computes the path from the origin to the destination node with a shortest path algorithm. The module then rents the intermediate location routes for a certain time window from the network manager. The time window for each location route is uniquely determined by the earliest departure time and the latest arrival time of the transportation task. The problem is now, that a location route in a railroad network cannot be used by two indepen-

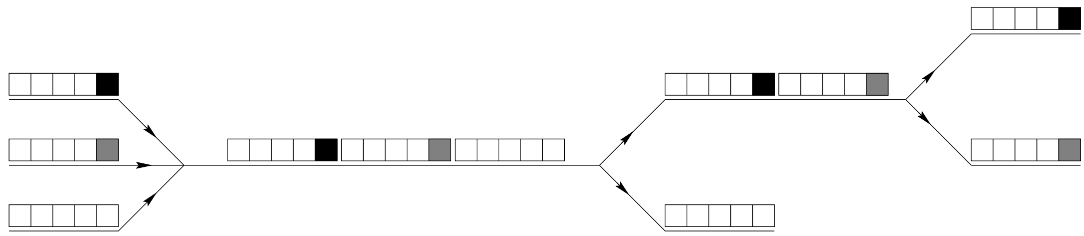


Figure 5.3: Train Coupling and Sharing (TCS)

dent modules at the same time. Either a route is blocked by a single module or two (or more) modules share a route by hooking together at the beginning of a location route and splitting up afterwards as shown in Figure 5.3. In order to use the underlying railroad infrastructure most efficiently, the railroad modules should share as many location routes as possible while taking care of their local constraints.

The main advantage of this approach is that it avoids a central planning authority that schedules all transportation modules. Instead, each module is autonomous and tries to achieve its goal, which is to deliver its freight to some destination node in the network. Thus, each module performs local optimization of the network throughput by sharing as many location routes with other modules as possible. The local optimization process of all modules eventually leads to a high, though usually only suboptimal, degree of resource efficiency. Besides this major advantage, a decentralized approach implies less coupling operations during the train composition process, a high degree of customer accessibility and lower costs because of the effective location route usage.

In this section, we have outlined the basic idea of the TCS approach for freight transportation in¡a railroad network. We will use this problem throughout the next sections to illustrate the basic ideas of each view and develop the fundamental design of the TCS/MAS system in the course of this chapter.

5.2 Task View

5.2.1 General

The goal of this first step in the MASSIVE method is to identify *what* the target system should do. We will therefore not think in terms of "agents" or "goals" because this could too easily restrict our view for other possible solutions. Ideally, the result of this step in the method should be the basis for the decision of the designer whether a multiagent approach is the best solution for the given problem or not.

Agents are kinds abstractions that should be viewed in a broader context then just under a functional point-of-view. The functional abstractions that are modeled in this step can spread over several agents or multiple functional abstractions can be encapsulated in a single agent. These decisions, however, are not the subject of this step that has the only goal to model the basic requirements of the target system.

5.2.1.1 Functional Requirements

Functional requirements are a specification of what the system should do. To obtain a first sketch of the functional requirements, it is often useful to analyze the intended workflow in the broadest sense. Thus, the designer should first outline the organizational context and the intended use of the target system. However, this analysis should not be too deep as it will be refined to much more detail in the Environment view discussed in Section 5.3. Instead, the analysis should yield an abstract definition of the Input and Output of the system and allow for a precise description of the system task. Even continuous systems can be described in terms of their impact (ie. transformations) on their environment and can thus be modeled by a functional approach. Mathematical formalizations are an extremely valuable tool in studying and describing the transformations performed by any software system and should be used whenever possible.

The initial mathematical model of the desired functional behavior is usually not complete nor is it detailed enough to allow for an immediate implementation of the system. Therefore, the model will go through a refinement process. To achieve a complete model of the desired behavior, several iterations of the entire development process are likely to be necessary because the complete model needs input from the other views as well as user feedback. The degree of detail, on the other hand, can be enlarged in a local enhancement cycle by performing a functional decomposition of the system task into several sub-tasks and by applying this stepwise refinement process recursively to these sub-tasks. Note that we avoid to talk of “goals” in this respect for the reasons explained in Section 5.2.1 because a goal implies the existence of some entity that pursues the goal and thus leads to an “agent-oriented view” through the back-door. The repeatedly applied functional decomposition eventually leads to a *task tree* that captures the fundamental functional aspects of the problem domain.

Figure 5.4 shows an example for a task tree. The tree describes some of the

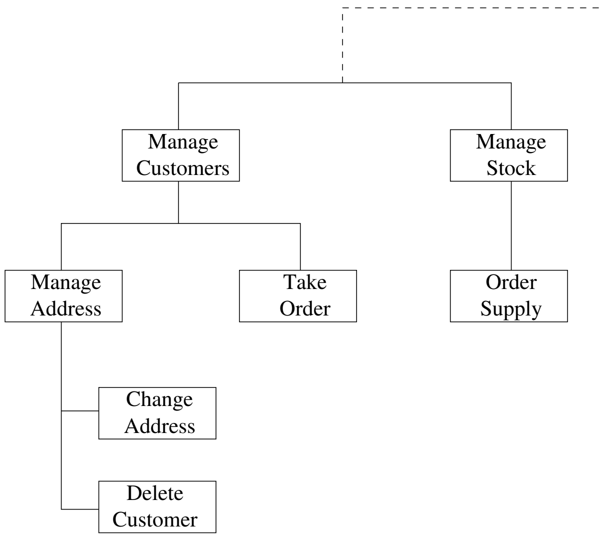


Figure 5.4: Example for a Task Tree

tasks that may occur in a mail order company that maintains a stock of goods and a database of customers that; the tree is embedded into a larger organizational structure indicated by the dashed line at the top of the figure. One can see in the figure how a particular task is decomposed into several subtasks, eg. the task "Manage Customer" is decomposed into "Manage Address" and "Take Order". These sub-tasks can be decomposed further as shown in the Figure.

The construction of the task tree follows the general idea of a hierarchical decomposition according to functional aspects of the problem domain. The granularity of the decomposition process depends on the specific problem but it should not become a specification of a particular algorithm. The task tree only provides a general picture of the activities that will be present in the target system, individual algorithms are designed later. The functional decomposition of tasks can be supported by the use of Structured Analysis (DEMARCO, 1978) or related methods.

Besides the functional modeling of the behavior of the target system, it is also important to quantify functional properties of the output (in the most general sense) of the system. Quantification means that the designer – ideally together with the user – must identify qualitative features of the solution that can be quantified by defining appropriate measures to decide that a solution generated by the systems satisfies particular requirements. Identifying the features that determine the quality of a solution and defined measures for these features is sometimes very difficult and should be done in close collaboration with the user to avoid the risk of designing a system that does not capture the users intention. This holds especially true in the case of optimization problems.

One of the most important rules for the functional modeling of the target system is to avoid any kind of control flow specifications. In this view, it is not important – sometimes even harmful – to think about *how* something has to be done. The goal of this view is solely to specify *what* is to be done. Furthermore, it should always be kept in mind that a complete functional specification is not possible in the first iteration and consequently, the initial model should not be overloaded by modeling features that show to be of less importance during the course of modeling the other views on the system.

5.2.1.2 Nonfunctional Requirements

The nonfunctional requirements define meta-level properties of the target system, ie. properties that usually specify additional constraints that cannot be modeled in the solely functional context. Typical examples for nonfunctional requirements are the stability of the system, security aspects, portability, extensibility etc. Nonfunctional requirements often affect a system as a whole. Because of their global nature, they should be specified only if it is absolutely necessary as they usually impose severe limits on the system design (ROMBACH, 1994A).

5.2.1.3 Basic Problem Solving Capabilities

In the course of defining the functional requirements of the target system, it is recommended to build a task hierarchy of the problem domain. Usually, this process leads to a task tree with the overall system task as the root of the tree that is decomposed into several sub-tasks. The leaves of the tree are either *atomic activities* that can be handled by a single problem solver or they are *joint activities* that require the collaboration of several problem solvers. The activities of the second group are handled in the Interaction view discussed in Section 5.5 whereas the activities of the first group are subject to this view.

Each of the atomic activities given in the task tree is specified on an algorithmic level that uses the abstractions of the mathematical model to describe a particular problem solving strategy. The basic capabilities that are defined in this view will later be combined and used as the basis for the role delimination process in the Role view.

5.2.2 Case Study

5.2.2.1 Functional Requirements

Workflow The workflow of a customer request for a transportation service according to the TCS approach discussed in Section 5.1 is shown in Figure 5.5. First of all, the service provider checks whether the request can be satisfied due to technical constraints (eg. if the customer requires a special wagon type, eg. cooled). If the required resources are available, the user request is made a so-called "task" and added to the task list of the service provider. A task is served by the service provider by finding a plan that satisfies additional user constraints such as freight availability or delivery deadlines and that also minimizes the cost for the service provider. If no plan can be found that satisfies these additional requirements, the task is rejected by the task dispatcher. If, on the over hand, a valid plan exists, the service provider must try to allocate the external resources that are needed to complete the task. The external resources in our case are the location routes that are used by the union that serves a particular task. If the resources can be allocated successfully, the task is finally accepted and scheduled for execution when the time is up.

Functional Requirements The focus of the TCS/MAS system in this scenario is the service provider which is given a list of task specifications that must be satisfied. The set of tasks is not fully known to the system at start-up time, new tasks arrive during the planning process and may require a revision of the already assembled plan in order to reduce cost. In order to have a more realistic simulation of the environment, the resource allocation process should be modeled as well. Furthermore, the system should be capable of simulating the resulting schedule as well as failures that occur during the execution of the schedule such as location route failures, ie. the case that an allocated resource is not available upon plan execution time and a process of dynamic re-planning must take place.

In Section 5.1, we have introduced the basic ideas of TCS. Recall that fundamental idea of TCS is location route sharing, ie. the idea that two or more modules hook together at the beginning of a location route (or of a sequence of consecutive routes) and split up afterwards.

For this case study, we shall use the seven node network shown in Figure 5.6 to illustrate the basic ideas of our approach. The numbers on the routes in Figure 5.6 indicate the distance between two nodes connected via a location route.

To illustrate the idea of route sharing, consider the following example with

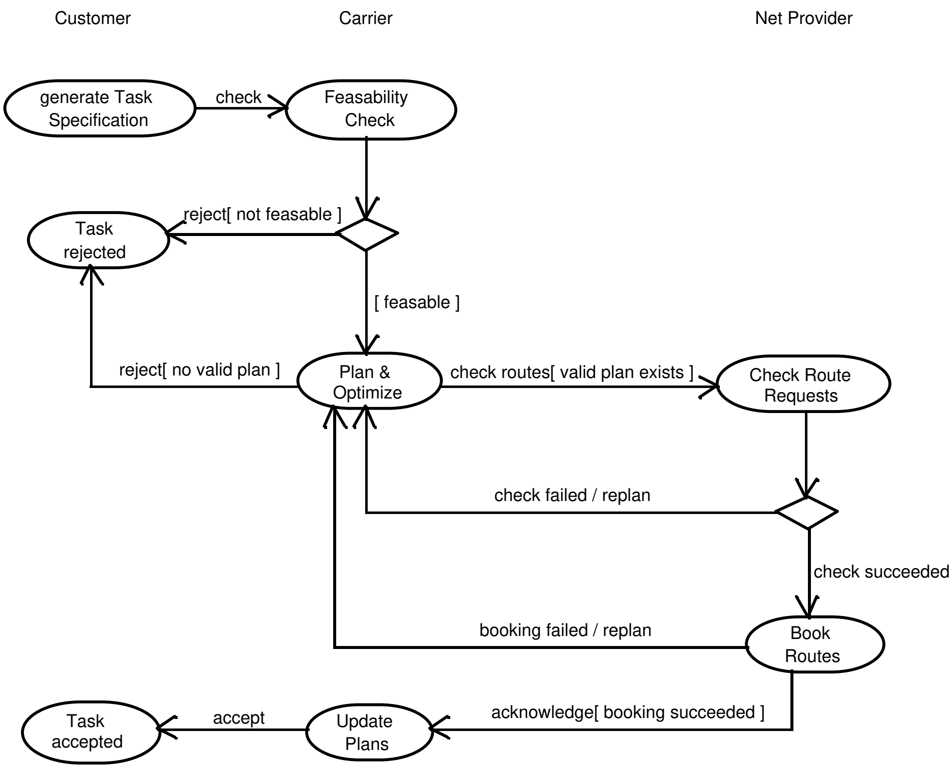


Figure 5.5: TCS Simulator Workflow (UML)

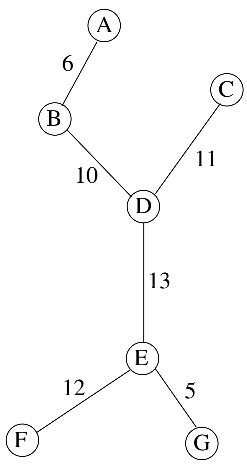


Figure 5.6: Example railroad network

two transportation tasks from B to F and from C to G, respectively. If the two modules serving the respective tasks act independently, the transportation costs for the first module are 35.0 units and 29.0 units for the second. If, on the other hand, the two modules decide to cooperate and to share the common location route between node D and node E and assuming that the cost for shared routes are equally distributed among the participation modules, the transportation cost reduces to 28.5 and 22.5, respectively.

The computational problem in conjunction with location route sharing is to identify sets of tasks (i.e. their respective modules) that can share location routes. The limiting factors are the compatibility of transportation paths and time windows. We will refer to sets of cooperating modules as *unions* where each union is determined by the participating modules.

The functional requirements are now informally given as follows:

- The system should be capable to plan the union formation process. Union formation means, that the system is able to find task specifications that can be combined into larger units according to the TCS approach.
- The system should be capable of dealing with an incomplete problem specification.
- The system should be able to simulate the execution of the resulting schedule.
- The system should be able to simulate failure situations such as the unavailability of a particular resource.

The task decomposition of the TCS/MAS system is shown in Figure 5.7. The activities in the leaf nodes of the task tree shown in the figure correspond to the basic problem solving capabilities that will be developed in Section 5.2.2.3, the double box indicates a leaf node that requires the collaboration of several problem solving entities and that is thus not subject to this modeling process. Joint capabilities will be dealt with in Section 5.5.2.

We will now construct a semi-formal model of the TCS domain that enables us to define the basic problem solving capabilities more precisely.

DEFINITION 5.2.1 (TIME WINDOW) *A* **time window** *TW is a tuple* $\langle \underline{t}, \overline{t} \rangle$ *that consists of a lower bound and an upper bound with* $\underline{t} \leq \overline{t}$.

DEFINITION 5.2.2 (PATH) *A* **path** *is a sequence* $\overline{A_1, \ldots, A_n}$ *of node identifiers such that there exists a link between any two nodes* A_i *and* A_{i+1} *for all* $i \in \{1, \ldots, n\}$.

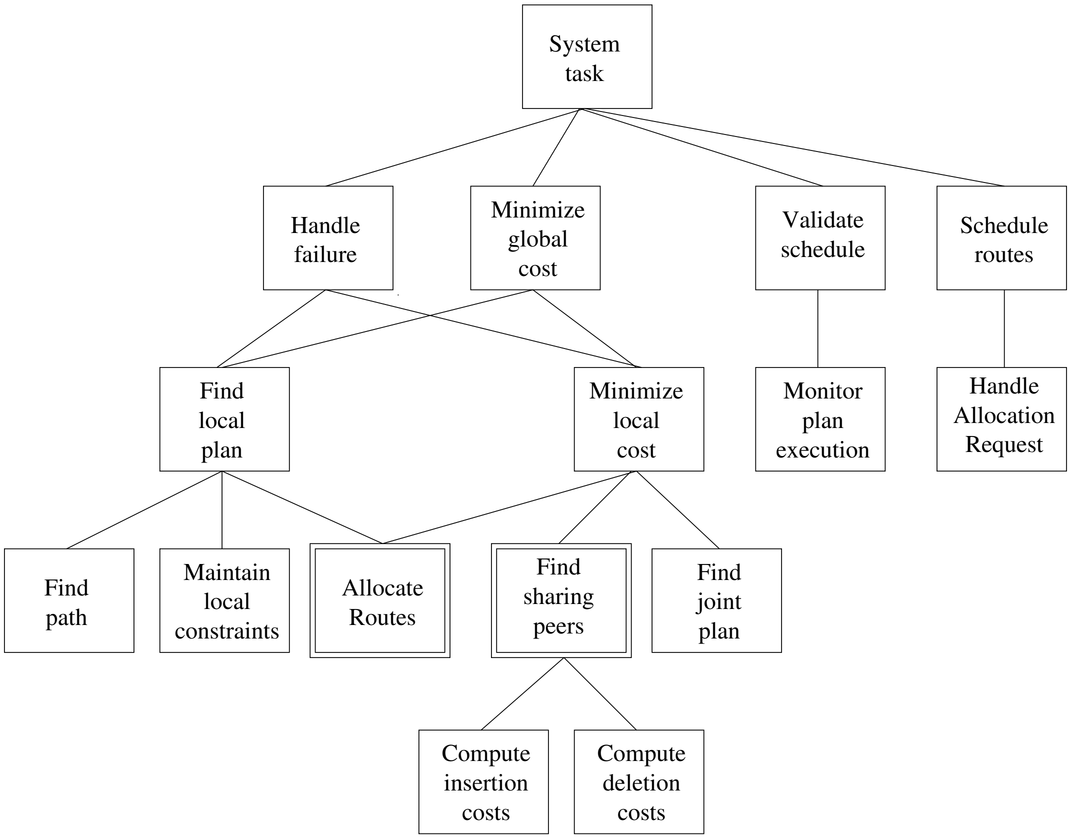


Figure 5.7: Task Tree of the TCS Domain

DEFINITION 5.2.3 (TASK) *A* **task** *is a tuple $\langle O, D, TW, AT\rangle$ consisting of the origin node O and the destination node D, a time window TW that specifies the earliest possible departure time and the latest allowed arrival time and finally a time stamp indicating when the task is announced to the system.*

DEFINITION 5.2.4 (MODULE) *A* **module** *M_i is specified by its unique identification number $i \in \{1, \ldots, m\}$ and a tuple $\langle P, S, L\rangle$ denoting its current plan P, the maximum speed S of the module and its length L.*

DEFINITION 5.2.5 (PLAN) *The* **plan** *of a module M is a sequence $[PS_1, \ldots, PS_n]$ of plan steps where each plan step PS_i is given by a tuple $\langle N, TW, A\rangle$ that consists of a node identifier N, a time window that specifies the earliest possible arrival time and the latest allowed departure time from the node and a list A of actions that must be executed in the node.*

DEFINITION 5.2.6 (ACTION) *An* **action** *is a tuple $\langle T, P, D, TW\rangle$ that specifies the* *action type T, the peer modules P, the action duration D and the time window during which the action must be executed.*

DEFINITION 5.2.7 (UNION) *A* **union** U_j *is a collection of several modules $M_{i_0}, \ldots M_{i_n}$. Each union has a unique identifier $j \in \{1, \ldots, k\}$ and is written as $U_j^{i_0,\ldots,i_n}$.*

Unions are-meta level concepts; a union emerges when at least two modules decide to cooperate by sharing location routes and it ceases to exists when all modules within the union have completed their respective tasks.

DEFINITION 5.2.8 (FAILURE SPECIFICATION) *A* **failure specification** *is a tuple $\langle L, TW \rangle$ consisting of the location route identifier L and a time window TW that specifies when the location route is blocked and when it can be used again.*

DEFINITION 5.2.9 (SCHEDULE) *A* **schedule** *is a sequence of tuples $\langle L, TW^* \rangle$ consisting of the location route identifier L and a sequence of time windows TW_i where each time window specifies when the location route is used by a train.*

The elements of the data structures are accessed using the dot notation scheme where the list elements are referenced by their index number. For example let

```
[
(NodeId:A, t̲:10, t̄:16, actions:nil)
(NodeId:C, t̲:10, t̄:16, actions:[(type:join peers:[M2 M3] duration:4 t̲:11 t̄:15) ])
(NodeId:D, t̲:10, t̄:16, actions:nil)
]
```

be the plan of module M. Then we have $M.A.\bar{t} = 16$ or $M.C.actions.1.peers = [M_2 \ M_3]$.

5.2.2.2 Nonfunctional Requirements

There are few nonfunctional requirements of the TCS/MAS system, the only one that must be obeyed is that the integration of a new task into an existing schedule should be within a time span of few minutes to allow the task operator to answer to a customer request quickly.

5.2.2.3 Basic Problem Solving Capabilities

In this section, we will develop the basic problem solving capabilities that are needed by the agents in the TCS scenario. The capabilities that are described in this section will later become the basis for the role delimination process for the roles of TCS agents.

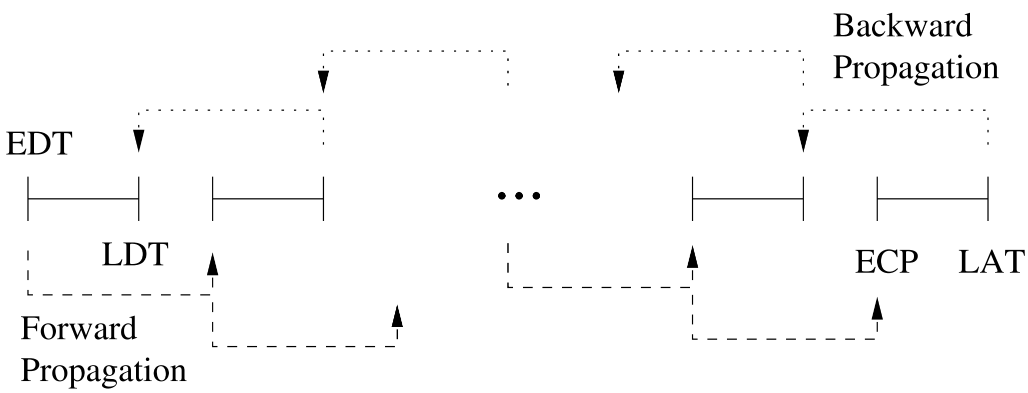


Figure 5.8: Time Window Propagation

Planing Algorithm for a single Task The planning process for a single task completion consists of three steps:

(1) Finding the shortest path between the source and the destination node.

(2) Forward propagation of arrival times.

(3) Backward propagation of departure times.

In the first step a simple Dijkstra-Algorithm is applied to the input nodes in order to find the shortest path between the source and the destination node. In the next step, the *buffer time* for the task is distributed over the intermediate nodes. The buffer time of a task is the fraction of the time window of the task that is not needed by driving operations of the module. Consider, for example, a task that is given by its source and destination nodes and its time window consisting of the Earliest possible Departure Time (EDT) and the Latest allowed Arrival Time (LAT). A module that serves this task will need a particular time to drive from the source node to the destination node. If the size of the time window of the task is bigger then the time needed to drive from the source to the destination node, the module has some additional time it can spent eg. because it must wait for a location route that is occupied by another module to become free again or if it wants to couple together with another module for location route sharing. The buffer time of the module is equally distributed over the intermediate nodes of the plan such that the module knows exactly for each node when it must arrive at the node and when it must leave from the node such that it can still fulfill the overall time window constraints.

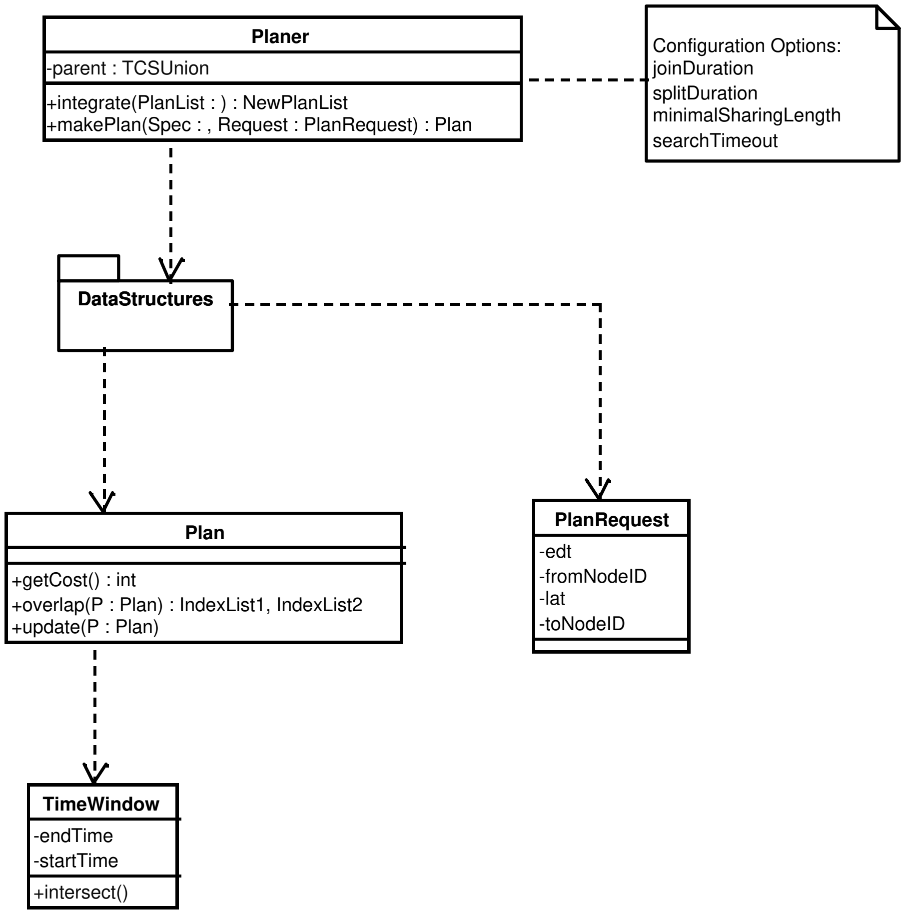


Figure 5.9: Planer (UML)

The process of forward and backward propagation is illustrated in Figure 5.8. If the module leaves the source node at time EDT then it will arrive (assuming that no route failures occur) at the destination node at time ECP (Earliest Completion Time). If, on the other hand, the module leaves at the Latest allowed Departure Time (LDT), it will not be able to complete its task before the LAT.

Plan Integration Operator We will now describe the plan integration operator for the plans of n distinct modules. Plan integration means, that the operator takes the n plans as input and modifies these plans by inserting *join* and *split* actions such that the resulting plans imply a maximum degree of location route sharing. The integration operator is used by the unions to decide whether they can integrate a new module into their set of modules or not.

We will use the following rather simple example to illustrate the basic ideas:

two unions U_1^1 and U_2^2 with modules M_1 and M_2 serving tasks $T_1\langle A, F, 10, 60, 0\rangle$ and $T_2\langle C, G, 9, 50, 0\rangle$ respectively. Basically, plan integration is achieved in five steps:

(1) **Find location route matches**
The first step in the plan integration operation is to find an overlapping sequence of location routes in the plans. In the example, the path of module M_1 is $\overline{ABDEF}$ and the path of module M_2 is $\overline{CDEG}$. Thus, the two paths overlap in $\overline{DE}$. If no overlapping is found, the plan integration process is aborted. For n plans, the general overlap condition is $\forall i \exists j : overlap(M_i, M_j) \neq \emptyset$, i.e. for each module, there must exist at least one sharing peer in the union.

(2) **Generate joint actions**
If the overlap condition holds, the next step in the plan integration process is to generate *join* and *split* actions for the respective plans. These actions are referred to as *joint actions* because they require two modules to coordinate their individual actions. In the example, the two modules join in node D and split in node E, the actions to be inserted are therefore

M_1`.D.actions =[(type:join peers:[`M_2`] ...)]`
M_2`.D.actions =[(type:join peers:[`M_1`] ...)]`

M_1`.E.actions = [(type:split peers:[`M_2`] ...)]`
M_2`.E.actions = [(type:split peers:[`M_1`] ...)]`

(3) **Minimize number of joint actions**
The number of joint actions generated in the previous step is not optimal because the generation process considers only the local context of the action, i.e. only a single step in the plan. Due to prior actions of a module, however, some actions are obsolete and can be eliminated. To illustrate this situation, assume another Module M with task $T\langle B, G, 10, 60, 0\rangle$. Integrating the three modules M_1, M_2 and M, yields three overlappings

(3.1) $(M_1, M_2) = \overline{DE}$

(3.2) $(M_1, M) = \overline{BDE}$

(3.3) $(M_2, M) = \overline{DEG}$

resulting in three bilateral actions pairs

(3.1) (M_1, M_2): *join* at D, *split* at E

(3.2) (M_1, M): *join* at B, *split* at E

(3.3) (M_2, M): *join* at D, *split* at G

This results in a generation of two *join* actions for module M_2 at node D: one with M_1 and one with M. These two actions can be reduced to a single action M_2*.D.actions = [(type:join peers:*$[M_1 M]$ *...)]* because M_1 and M are already linked due to their prior *join* operation at node B. While this is rather trivial in this example, it is not the case in more complex plans where previous *join* and *split* actions must be recursively traced for a large number of modules.

(4) **Specify joint action constraints**
In this step, the time windows of the newly generated actions are specified. The conditions that must hold are that actions must take place within the time windows of the plan steps of the respective modules and that the joint actions must take place simultaneously. In the example, the resulting constraints are
$M_1.D.actions.1.\underline{t} = M_2.D.actions.1.\underline{t} = max(M_1.D.\underline{t}, M_2.D.\underline{t})$
$M_1.D.actions.1.\overline{t} = M_2.D.actions.1.\overline{t} = min(M_1.D.\overline{t}, M_2.D.\overline{t})$

(5) **Find action schedule**
In the last step of the plan integration process, the operator must guarantee the existence of a schedule for all actions in each plan step of each module. This means, that all actions occurring in a plan step must be serialized in a way that the action executions do not overlap in time.

For example let M_i`.N.actions = [(type:join peers:[`M_j`] duration:4` $\underline{t}$`:10` $\overline{t}$`:30) (type:join peers:[`M_k`] duration:4` $\underline{t}$`:14` $\overline{t}$`:34)]`

be the plan step of module M_i at node N. The time windows for the two actions are shown in Figure 5.10, the task of this step in the plan integration process is to arrange these actions within their respective time windows such that they do not overlap and that they take place in the time interval given by cutting the action execution time intervals. A valid schedule for the two actions is also shown in Figure 5.10.

As one can see in Figure 5.10, there are multiple schedules for the actions exist (eg. action 1 could as well be scheduled after action 2). However, the time windows of the actions are not committed to a particular schedule because this implies an unnecessary restriction imposed at planning time. At planning time, it is sufficient to know that a schedule exists, concrete commitments to a particular schedule are delayed until execution time giving the participating modules a maximum degree of flexibility.

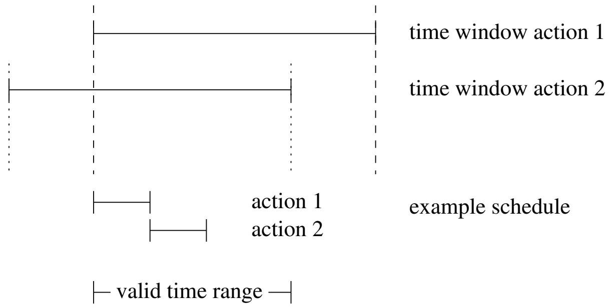


Figure 5.10: Action Scheduling

After these steps of the plan integration process have been successfully performed, the output of the plan integration operator is a list of module plans that satisfy the overlap condition mentioned before.

Decision Functions In order to decide whether to integrate a new module into an existing union or to exchange a module in the union with an external module, the unions must have decision functions that determine their optimal behavior.

The *insertion cost* c^+ of a module m into an existing union U is

$$c^+(U,m) = \sum_{r \in U'.routes \wedge |r.users|=1} \omega(r) \quad (5.1)$$

where U' is the is the union that emerges when the plan integration operator presented in the previous section is applied to the plans of the modules in U and module m. $U'.routes$ denote the location routes used by U' and $|r.users|$ is the number of modules using location route r simultaneously. Thus, the insertion cost are the additional costs that occur because of location routes that are not already used by the union.

The *deletion costs* (or *savings*) c^-, on the other hand, occur when a module m is deleted from a union U and compute to

$$c^-(U,m) = \sum_{r \in m.routes \wedge |r.users|=1} \omega(r) \quad (5.2)$$

ie. the savings are the costs of the routes of module m that are not shared with other modules.

Thus, it pays to switch a module from an union U_1 to another union U_2 if the insertion costs for the module in U_1 are less than the costs that occur when the module is deleted from union U_2, ie. $c^+(U_2,m) < c^-(U_1,m)$.

Plan Execution Simulation The plan execution monitor (PEM) is the link between the planning unit and the real world (or a simulation engine that simulates the real world). The PEM controls the usage of location routes and the coupling activities of the modules. To illustrate how plan execution monitoring works, recall the plan of module M_2 from the example in the previous section:

[
(NodeId:C, $\underline{t}$:9, $\overline{t}$:21, actions:nil)
(NodeId:D, $\underline{t}$:20, $\overline{t}$:32, actions:[(type:join peers:[M_1] duration:4 $\underline{t}$:11 $\overline{t}$:15)])
(NodeId:E, $\underline{t}$:33, $\overline{t}$:45, actions:[(type:split peers:[M_1] duration:3 $\underline{t}$:11 $\overline{t}$:15)])
(NodeId:G, $\underline{t}$:38, $\overline{t}$:50, actions:nil)
]

When the module starts to execute this plan at time $t = 9$, it asks the PEM whether the location route from C to D is available as it should be because it was allocated at planning time. However, the route may not be available at execution time due to external reasons, e.g. route blocking due to mechanical failure of another module while using that particular route and it is the task of the PEM to check whether or not the route is available. If the module is allowed to use the route, it departs from node C. When it arrives at node D, it issues a coupling request to the PEM, indicating that it is waiting for module M_1 to join with it. If module M_1 has already arrived at node D, the coupling action can start if M_1 is not engaged in an ongoing coupling activity with another module. If M_1 has not arrived yet or is currently engaged, the coupling action of M_2 is stalled. If M_1 does not become available within the time interval specified in the *join* action of M_2, M_2 departs from D without coupling with M_1 because otherwise M_2 will miss its scheduled latest arrival time at the goal node. If the coupling activity can be completed as it is scheduled, the two modules depart from D after the coupling is finished. When they arrive at node E, the two modules inform the PEM that they want to split and after completing the *split* action they depart from node E for their respective goal nodes.

This example is, again, highly simplified. If more than two modules are involved in *join* or *split* actions additional plan integrity constraints must be satisfied. If, for example, three modules $M_1M_2M_3$ are coupled in this order and M_2 must split from M_1 and M_3, an additional *join* action between M_1 and M_3 must be generated because M_1 and M_3 are supposed to remain coupled. However, no additional *join* action is necessary if the original module order is $M_2M_1M_3$. The decision of whether to generate additional coupling actions or not must be taken by the PEM upon plan execution time, depending on the actual coupling order of the modules.

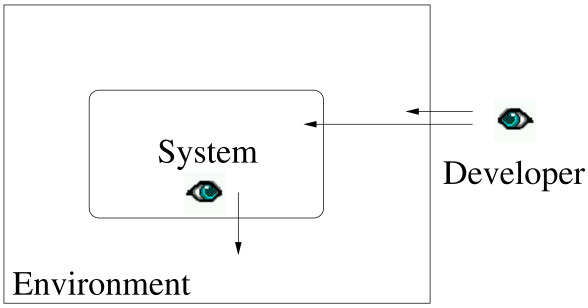


Figure 5.11: Perspectives on the Environment

5.3 Environment View

5.3.1 General

The goal of this view is to model the environment of the target system from two distinct points-of-view as shown in Figure 5.11. Firstly, the environment will be analyzed and a general characterization of the environment as it appears to the system developer will be generated. Secondly, the environment will be described from the systems point-of-view in terms of input/output or sensor/effector specifications.

5.3.1.1 Developers Perspective

This part of the Environment view describes the environment of the multiagent system from the developers point of view. The main aspects are the organizational context in which the system will be used, the general characteristics of this context as far as it is relevant for the application and a technical assessment of the runtime environment of the operational system.

Organizational Context The first step in modeling the environment of the target system is to clarify the organizational context. In the Task view (Section 5.2), this aspect has been addressed in a rather superficial manner and it is now time to refine the abstract initial model to a more specific description.

The organizational context determines how and where the target system should be used within the customers organization. If, for example, the target system is meant as a prototypical evaluation of a new approach or some other technology that will not be used in an operational context, it surely directs the focus of the final design onto more functional aspects then onto nonfunctional aspects such as stability or user friendliness.

It is hard – if not impossible – to specify concrete rules of how to specify the organizational context because the relevant concepts are extremely fuzzy. Therefore, the major contribution of organizational context specification in this view is to provide the designer with a feeling of how the system should be used and what is important with respect to the field of application of the target system.

Characterization (RUSSELL AND NORVIG, 1995) propose a very general scheme to describe the environment of agent-based applications that uses five dimensions to capture the characteristic features:

- *accessible* vs. *inaccessible*
  In an accessible environment, the agents can perceive the full state of the environment at any time whereas in a an inaccessible environment only partial information is available to the agents.

- *deterministic* vs. *nondeterministic*
  This feature describes how the agents perceives the development of its environment, ie. whether the next state of the environment is completely determined by the current state (as perceived by the agent) or not.

- *episodic* vs. *nonepisodic*
  An episode is a single perception-action cycle performed by the agent. In an episodic environment, these cycles are unrelated, ie. the action of one cycle does not have any impact on the next cycle. Thus, episodic environments are much simpler then nonepisodic environments because they do not need the ability of the agent to plan ahead.

- *static* vs. *dynamic*
  In a static environment, the environment cannot change while the agent is not acting (ie. while the agent is deliberating) whereas this assumption does not hold in a dynamic environment.

- *discrete* vs. *continuous*
  In a discrete environment, the agent has distinct, clearly defined percepts that (partially) describe the environment. This is not the case in a continuous environment.

These features can be used to characterize a broad range of environments for example the physical world is inaccessible, nondeterministic, nonepisodic, dynamic and continuous, whereas a virtual world is also nondeterministic, nonepisodic and dynamic but it is usually accessible and discrete.

Characterizing the environment according to the above features can help the designer to anticipate basic problems that are likely to occur in a particular class of environments. Dynamic environments, for example, require a much more sophisticated reasoning mechanism that is capable of integrating newly perceived state information in the ongoing reasoning process.

Runtime Environment The next important aspects to consider deal with technical aspects of the runtime environment of the target system. These characteristics are often neglected because they are considered unimportant and could therefore be left out without loosing generality. But this is not true! The runtime environment can have an enormous impact on he final design and performance of the system and should thus be explicitly modeled as early as possible. A good example for the impact of the runtime environment on the final system is the TEAMWORK LIBRARY discussed in Section 6.2. The original approach used simulated broadcast because physical broadcast was not offered by the communication platform and it was assumed that simulated broadcast would do as well. This turned out to be a mistake because the performance and value of the teamwork approach relied on the fast transfer of large amounts of data. Thus the entire system had to undergo a re-design phase that added mechanisms and protocols for physical broadcast. To avoid such expensive error-corrective work, the runtime environment must be carefully characterized and evaluated.

First of all, the designer must specify the programming model that should be used in the target system. Programming models can be classified according to their degree of concurrency: a *sequential* programming model admits for no concurrency at all; functions are called strictly one after the other. This kind of programming model does not require synchronization mechanisms and it is sufficient in case the multiagent paradigm is solely used for design purposes, ie. to structure the target system. A *pseudo-parallel* model, on the other hand, typically uses light weight processes (threads) within a single operating system process to achieve concurrent execution of program fragments. This kind of programming model is well suited for multiagent applications because it allows for a reasonable degree of parallelism while still being easy to handle wrt. debugging and traceability. A fully *distributed* programming model, finally, distributes the computation space over several computers connected via a network. This model is quite hard to handle because it often requires complex synchronization and fault tolerance mechanisms. However, for real world applications, this might be the only possible choice.

In the next step, the designer has to describe the communication platform that will be used within the target system. The choices for the communication platform are usually limited by the choice of the programming model; a sequential system, for example, will usually be limited to internal communication using ordinary method invocation mechanisms. In the case of a pseudo-sequential programming model, the choices are broader because such a programming model usually requires asynchronous message exchange to exploit the (pseudo-)parallel threads of execution. Asynchronous message invocation methods are usually not offered by the programming language and thus the designer must define a method that is best suited for communication between the entities of the target system. The widest range of choices, however, is given in the case of fully distributed systems. The entities can communicate using network protocols such as TCP/IP or radio transmission protocols such as GSM to exchange messages.

Finally, the choice of the appropriate programming language is an important issue. The programming language need not support a particular programming paradigm, eg. object-oriented programming although a cleaner, more structured code will be generated if the programming language supports particular features that are characteristic for multiagent systems.

5.3.1.2 Systems Perspective

The developers perspective of the system, however, is only one side of the coin. Specific parts of the system may have a very different perception of the environment given their physical or conceptual constraints. Usually, the system (and later the agents) perceive their environment through a number of sensors and operate on the environment by using effectors. First of all, this very general concept of an input/output specification must be clarified with respect to the characteristic needs and requirements of the target system and secondly, the designer must develop a model of the environment that is internally used by the agents to represent their environment and to reason about it.

The definition of the mechanisms that enable the entities in the system to perceive and act upon their environment should be general enough not limit the choice of assigning particular sensors and effectors to the roles within the multiagent system defined later (see Section 5.4). In this view, only a general model of the sensors and effectors that allows the designer to get an overview of how the systems inter-operability with its environment can be arranged should be generated.

The model of the sensors and the specification of the individual percepts then leads to the choice of the data structures that are needed to capture an internal world model. The internal model depends on the functional requirements of the target system. Sometimes, an internal model is completely obsolete (BROOKS, 1991) whereas in other cases, the agents must be equipped with a rather fine-grained model of their environment and their fellow agents.

5.3.2 Case Study

5.3.2.1 Developers Perspective

In order to characterize the development context of the TCS/MAS system, we will at first analyze the customers ideas of what the system should be used for. According to the Deutsche Bahn AG, the TCS/MAS system has two major goals:

Firstly, it is a strategic tool to evaluate the basic characteristics of Train Coupling- and Sharing. "Strategic" means, that the system is not intended to be used in an operational context in order to plan and organize freight transport according to the TCS approach. Instead, the system should be used to identify the relevant parameters and their optimal settings according to a given set of quality measures (eg. cost of a particular schedule or average number of coupling activities).

Secondly, the TCS/MAS system was developed to evaluate the technology behind it, namely multiagent problem solving. This evaluation is done from the functional point-of-view, ie. the customer wants to find out whether multiagent technology can solve the TCS specific scheduling problem or not. From the non-functional point-of-view, the customer is also interested in the question of how good the multiagent solution is. The quality of the solution depends not only on functional measures such as overall cost, but also on nonfunctional quantities such as response time to a new task.

The necessity to evaluate the TCS approach in general results in many constraints towards the organizational context of the TCS/MAS system because the system is required to obey the most important railroad operation regulations. These regulations affect the network specification of the underlying railroad network as well as specifications for coupling activities, maximum train lengths or maximum train speeds. A special case is resource allocation within the TCS/MAS system. The resources that must be scheduled among the competing modules are the location routes between any two nodes in the network. For these location routes, a predefined schedule exists that must not be violated by the resource assignment algorithm. Furthermore, the assignment algorithm must respect the minimal distance that two trains must keep when they are using the same location route.

Using the characterization scheme given in section 5.3.1, we can classify the environment as

- nonaccessible because no single entity has a complete model of the entire system,
- nondeterministic because of the incomplete view of any entity, the entity is not able to predict the next system state,
- nonepisodic because the actions performed in one cycle (forming of unions out of modules) affects the possible actions of the next cycle,
- static because the tasks are integrated one after the other and, finally
- discrete because of the granularity of the planning process.

The technical development environment features a pseudo-parallel programming model because it offers a reasonable degree of flexibility and concurrency while still being relatively easy to monitor and debug. The chosen programming language is a functional, constraint based, multi-threaded language called Mozart (PROGRAMMING SYSTEMS LAB, 1999), (SMOLKA, 1995) and the communication platform uses Mozart threads to achieve asynchronous message exchange between the agents. The system platform is Linux.

5.3.2.2 Systems Perspective

The perception and action possibilities of the system (and later the agents) are rather limited as it is often the case with systems of communication software agents. The agents can perceive all other agents that are currently active within the system and the only action that is allowed for an agent is to send a message to another agent. The knowledge structures that are needed to model the agents knowledge about the environment are therefore very simple and consist only of a list of peer agents that are currently active in the system.

5.4 Role View

5.4.1 General

Up to now, the tasks of the target system and a characterization of the environment where the system will be running after it is developed have been specified without committing to a particular technology to solve the given task. In this section, we will now look at this and start with modeling the system according to the multiagent programming paradigm. We will firstly define the concept of "role" by reviewing some existing characterizations and role modeling techniques and then present our recommendations.

However, in this section we do not only discuss the problem of finding role abstractions given the functional and physical constraints of the problem specification, but we will also look at the problem of how to assign particular role(s) to agents of the target system.

5.4.1.1 Role Definition

We have already mentioned in Section 2.1.2 that a major problem in the field of sociology is the delimination of roles that occur within a society.In the design of a multiagent system, the designer is faced with the problem of role delimination as well. He or she is given a functional specification and it is their task to structure this functional specification to form a society of interacting roles.

There are three potential ways to achieve this goal. The first approach is the classical *top-down* processing of the functional specifications (BARBUCEAN AND FOX, 1995). The specifications are decomposed into hierarchical sub-groups and these groups are then declared as the roles. This approach has the advantage that it is straightforward and that it usually leads to satisfactory results. Unfortunately, the approach assumes that the designer has the full flexibility in grouping the functionally decomposed units. However, this assumption does not hold for example in the case of a robotic application where the physical entities limit or even define the functional grouping.

Therefore, the designer may decide to go the other way round, namely to follow a *bottom-up* approach to identify the roles within the system. This approach (PARUNAK, 1997), (PARUNAK, 1999) starts from the physical grouping that is usually determined by the environmental context of the target system or by the physical capabilities of the existing entities and tries to create the interactions that fulfill the functional requirements. The idea outlined in (PARUNAK, 1999) is to apply linguistic case theory analysis on an informal task description as it is eg. given by the Task view (Section 5.2). First of all, Linguistic Case Theory (COOK, 1979), (PARUNAK, 1995) assigns a set of slots to each entity in the problem domain. These slots are typically filled with nouns associated to each verb in the informal description and each slot reflects the semantic connection of the role inhibitor to others and the verb and the collected nouns then become potential candidates for role descriptions. Although this approach is in accordance with the organizational or physical constraints of the target system, in its pure form it is only applicable if such a physical grouping really exists.

In the MASSIVE design method, we have tried to combine the advantages of

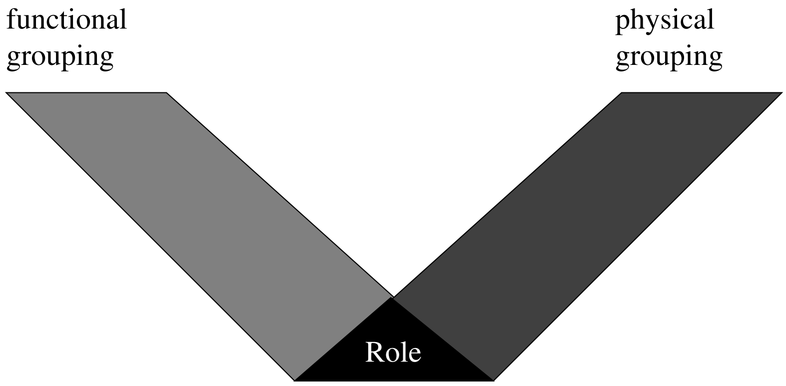


Figure 5.12: Hybrid Role Identification

these two approaches and to avoid their limitations as far as possible. First of all, we need a definition of what we see as a role.

DEFINITION 5.4.1 (ROLE) *A* **role** *is a logical grouping of atomic activities according to the physical constraints of the operational environment of the target system.*

Definition 5.4.1 tries to combine the top-down and the bottom-up approach towards role delimination by equally weighting the existing functional and physical groupings. Figure 5.12 illustrates the basic idea of our *hybrid strategy*: the functional decomposition and the physical structure are integrated into a common role that fulfills the functional as well as the physical requirements. The resulting micro process is pretty much straightforward as shown in Process Model 4.

Besides these *functional roles*, we have a second group of roles in a multiagent system called *interaction roles*. Interaction roles are prototypical roles that are modeled in interaction protocols and are not explicitly defined by the system designer. They are determined by the interaction protocols that are used within the multiagent system, eg. when the designer decides to use the Contract-Net protocol (SMITH, 1980), the generic interaction roles *manager* and *bidder* must be implemented according to the particular context. Interaction roles, – despite of their name – however, are not a genuine part of the role view and are therefore discussed in the Interaction view in Section 5.5.1.

Process Model 4 MASSIVE Role Modeling

(1) **Identify functional groupings**
In this step, the basic capabilities are grouped together into functional roles such that each role represents a coherent cluster of functions that is sufficiently different from other clusters. (COLLINS AND NDUMU, 1998) suggest two rules of thumb that can support this process:

- The *Sphere of responsibility test* aims at identifying local resources that are used by a group of functions and to associate this group with a particular role.
- The *Point of interaction test*, on the other hand, is used to separate the different functional groupings from each other in order to achieve a high degree of partitioning of the target system.

Thus, the two rules of thumb clearly relate to the concepts of coupling and cohesion.

(2) **Identify physical groupings**
In the second step, the physical constraints that are present in the problem domain are identified and documented. As we have already stated above, the physical groupings are usually determined by the system environment eg. hardware devices or geographical constraints.

(3) **Integrate**
In the final step of this micro process, the functional roles are unified with the physical entities. It is important for the designer to define a mapping from one group onto the other that achieves a high degree of resource efficiency ie. that does not overload a particular resource while others are not used to their possible extend. If, for example, a workstation represents a physical role, the functional roles that are mapped onto this workstation should not be of a computational complexity that cannot be handled by a single computer nor should they be so lightweight that the machine is idle most of its time.

5.4.1.2 Role Assignment

Role assignment is the process of mapping roles as they were introduced in Section 2.1.2 onto agents. By defining agents as the sum of the agent architecture and the potential roles the agent can play as we did in Section 2.1.4, we can state the problems within the role assignment process by proposing two major questions:

(1) How can the roles be mapped to agents?

(2) How are general roles mapped to interaction roles?

A role assignment within a multiagent system can either be *static*, ie. the agent is assigned a particular role at system start-up or it can by *dynamic*, ie.

the role assignment to an agent can change during the agents lifetime. Furthermore, an agent can play its different roles sequentially or it can have several roles at the same time.

Static role assignment has the advantage that the system designer need not think about how the agents internal state changes during a role change can be modeled. The agent is bound to a fixed task and no intra-agent conflicts of any kind can occur. However, static role assignment is not feasible if different roles share particular resources, for example cognitive structures. The knowledge transfer from one agent to another can be prohibitive for a static role assignment and so a dynamic scheme must be applied. An example for this case is discussed in Section 6.2.

Dynamic role assignment becomes a very difficult problem if all agents have the same capabilities and the role assignment process should be arranged by the agents themselves. In (FISCHER, 1993) this problem is discussed in depth and a generic interaction protocol for the dynamic assignment process is presented.

5.4.2 Case Study

In this section, we will develop the Role View for the case study according to the hybrid role delimination process presented in Section 5.4.1.1. We will at first perform the functional grouping of the task decomposition given in Section 5.2.2 and then we will briefly discuss the physical grouping that pre-exist in the problem domain. In the next step, we will then aggregate these two groupings in order to find the final role models.

5.4.2.1 Functional Grouping

TCS Module The task of a *TCS module* (or simply module) is to transport goods within a railroad network. To achieve this task, the module has to perform a local planning process that yields a plan through the railroad network and that satisfies the customer constraints with respect to freight availability and freight delivery deadlines.

TCS Union As already mentioned in Section 5.2.2, a *TCS union* is a meta-level structure that emerges whenever two or more modules decide to share a particular location route. A union integrate additional modules if there are sharing possibilities with the union members and thus reduce its costs by performing a local optimization process. Furthermore, a union can also release one or more of its members from the union if this allows the union to integrate another module thats cost saving potential is higher then the saving that was gained with the original module. Global optimization within the society of unions emerges from the local optimization steps of the individual unions.

Task Manager The *task manager* is responsible for handling incoming transportation requests. The task descriptions are externally specified; each task has a time stamp that indicates, when (i.e. in which clock cycle) the task should be announced to the system. This mechanism allows to simulate time varying task arrival-times.

Net Manager The *net manager* controls the location route usage by serving the route allocation requests of the unions. It has direct access to the underlying railroad network. The major task of the net manager is to guarantee that no location route is allocated to more than one module or union during a particular time window. Furthermore, the net manager is responsible for managing an existing schedule on the location routes (i.e. passenger trains) and thus limit the available time windows for some location routes.

5.4.2.2 Physical Grouping

The physical structure of the problem domain is not necessarily important for the structure of the target system because the system implements a virtual multiagent system that is running in a single process.

In future extensions of the TCS/MAS to a fully operational system, however, there is the possibility to distribute the agents to physical entities, for example the transportation modules that handle the transportation tasks or a remote machine that handles the resource management process. Thus, the design of the multiagent solution should reflect this possible extension and be flexible enough to serve as a basis for a transfer of the software architecture to the physical world.

5.4.2.3 Aggregation

The functional and the quasi nonexistent physical groupings presented in the previous paragraphs suggest that it is quite natural to use the functional decomposition as the final role models. This natural practice is justified in the case of the task manager and in the case of the net manager as these roles are independent of any other functional role and so they can be modeled in isolation. However, this is not the case with the modules and unions. These roles have a rather tight coupling and this coupling should be reflected in the system design.

We have assumed so far that a union emerges when two or more modules decide to share a location route. Even though this view reflects the basic idea of the TCS approach, it is not really appropriate in the design of the software system that implements the approach. For a more clearly structured design, it is a better idea to view each individual module as a special case of a union – simply a union with a single module, in the context of this work referred to as a *degenerated union*. Applying this scheme results in a much simpler system design because it allows us to treat modules and unions uniformly. Thus, we have less abstractions and less complexity in our design. Abstractions, on the other hand, are of course extremely valuable when a problem must be decomposed into several parts. But this example demonstrates that an extensive use of abstractions can lead to difficulties that can be circumvented if some abstractions are merged together into larger units.

However, the unions are not the only functional roles in the TCS/MAS system as we also have the task manager and the net manager that controls the resource handling process. As described above, the net manager is responsible for accepting incoming location route requests and computing a schedule that uses the underlying resources most efficiently. Because of the fact that there are no other related tasks in the system, the entire functional role is determined by a single activity. This holds also for the task manager which does not have related functional aspects and is thus also mapped in a one-to-one manner to the functional role.

5.4.2.4 Role assignment

In the TCS/MAS system, a static the role assignment scheme is applied because of the natural mapping from the functional roles to agents. There is no need for an agent to switch between functional roles in the application and thus the role assignment can be hard-coded into the implementation.

5.5 Interaction View

5.5.1 General

The roles that have been developed in the previous section are fundamental abstractions that decouple the functional aspects of the system from the agents. However, some functional aspects require the collaboration of several agents in order to be fulfilled and these collaborations must be specified by the system designer in one way or another.

In this section, we will characterize some fundamental types of interaction in their most general form and then provide some guidelines how these generic forms of interaction can be used in a multiagent system. A focus of this section will be the explicit communication between agents as it is by far the most common form of interaction implemented in the multiagent systems found in the literature.

5.5.1.1 Conflict Resolution

Coordinated interaction among several autonomous entities is the core concept of multiagent technology. In this section, we will briefly characterize the major forms of interaction and illuminate the potential difficulties in the interaction design process. However, we cannot even try to provide a thorough overview over the topic because the issues related to this thematic complex are just too broad. We will therefore limit ourselves to some basic concepts and ideas and direct the reader to the specific literature whenever it is appropriate.

Coordination in a (natural or artificial) society is the process of conflict resolution within the society and can be achieved in a number of ways. The most natural conflict resolution strategy that can be found in a physical environment is simply to do nothing. The laws of physics clearly define the outcome of actions that include more than one agent. For example, two robots that approach the same location will be coordinated by the physical law that only one of them can occupy the particular location. Thus, either the first robot to reach the location or the stronger of the two robots will finally occupy it. Obviously, this is a somewhat artificial example for a coordination strategy and it far away from being reasonable. Although it is straightforward and therefore easy to implement, it is usually a better idea to implement some sort of collision avoidance strategy except for the case when you have extremely robust robots.

The second conflict resolution strategy uses external mediation to solve the conflict (GEORGEFF, 1983), (CAMMARATA ET AL., 1983). Mediation means that the conflicting parties apply to a third, neutral party that decides what should be done. The most important prerequisite for this sort of conflict resolution is the mutual agreement of the agents to obey the decision of the mediator. The advantage of the mediation solution is that the decision about what to do is not based on local preferences of the agents but on a more global view (depending on the knowledge of the mediator). However, this sort of conflict resolution is often not seen as "real" multiagent technology because the agents loose some of their autonomy by relying on an external mediator.

The third way of conflict resolution, finally, is negotiation. This approach is mostly used in multiagent systems and it has shown to be a powerful tool to solve all kinds of conflict situations. In a conflict resolution process based on negotiation mechanisms, the agents exchange messages until they have reached a agreement on how the conflict is settled to their mutual benefit.

Besides its ability to solve conflicts among agents, negotiation mechanisms are a good means to attack complex optimization problems by simulation a market situation where the agents negotiate in order to find a solution that optimizes the local performance of the agents as well as the global performance of the entire agent society.

Coordinated interaction is the core concept of multiagent systems and the most common form of coordinated interaction that is used in multiagent systems is negotiation. Therefore, we will now take a closer look at negotiation processes as general problem solving mechanisms.

In order to negotiate, the first thing to organize is that the agents are able to get in contact with one another. This requires some sort of agent framework that manages the initial contact between two or more agents. Currently, several international companies and research institutes are working on the definition of an industrial standard in this area ((FIPA, 1997), (FIPA, 1998), see also Chapter 6.1) which provides a uniform framework for agent and multiagent applications.

Besides getting in contact with each other, the agents must also be able to understand each other. Several agent communication languages have been proposed (MACGREGOR AND BATES, 1987),(BUSSMANN AND MÜLLER, 1993), (FIPA, 1996), but it seems that the KQML (Knowledge Query and Manipulation Language, (FININ AND FRITZSON, 1994), (MAYFIELD ET AL., 1995)) has become a de-facto standard as agent communication language.

The goal of the KQML project is to develop a method for distributing information among different systems (NECHES ET AL., 1991) that clearly separate the semantics of the domain-independent protocol from the semantics of the domain-dependent messages. KQML messages have a fixed structure that allows each receiving agent to understand the message structure. To understand the content of the message, the agent must have additional knowledge.

The basic structure of any KQML message is the following:

```
(<KQML-Performative>
                    :sender   <word>
                    :receiver <word>
                    :language <word>
                    :ontology <word>
                    :content  <expression>
                    ...)
```

Each message consists at least of a so-called *performative*. Performatives were introduced in speech act theory (SEARLE, 1969) to describe the illocutionary force (ie. the intended meaning) of an utterance. Speech act theory was discovered by multiagent researchers as a valuable tool to describe the meaning of messages that are exchanged between agents (WERNER, 1988).

The structure of a KQML message is divided into several fields that have fixed semantics. In the above example, the `:language`, `:ontology` and `:content` field define the language in which the message is expressed, the vocabulary of the language and the message itself, respectively. The content language can be anything from user defined data structures up to LISP or PROLOG Programs or SQL queries. One of the most common content languages is the Knowledge Interchange Format (KIF) (GENESERETH AND FIKES, 1992).

KQML messages or any other types of messages define single messages that are exchanged between the agents, a more complicated matter is to organize the orderly exchange of sequences of messages. A common technique to describe the course of a message exchange process is to use protocols. Basically, a protocol defines which messages can be exchanged, the intended order of the message and possible or expected reactions of the participants to incoming messages. Furthermore, a protocol can also specify what to do in unexpected situations such as message loss or other error situations. A number of protocol specification languages have been proposed ranging from specification languages for low level communication protocols (THE INTERNATIONAL ORGANIZATION FOR STANDARDIZATION, 1997), (HOLZMANN, 1991) up to high level specification languages for multiagent applications (BURMEISTER ET AL., 1995), (KOLB, 1995). In the next section, we will present a protocol specification system that was developed to support protocol design, implementation and reuse.

5.5.1.2 Protoz: A Protocol Compiler

Protoz (PHILIPPS, 1998), (PHILIPPS AND LIND, 1999) is a protocol specification system that was developed in the Multiagent Systems Group of the DFKI. Protoz was designed to be as simple as possible but still powerful enough to capture the most relevant negotiation protocols currently in use. The specification language is related to Estelle (THE INTERNATIONAL ORGANIZATION FOR STANDARDIZATION, 1997) and uses a similar computational concept. Furthermore, the Protoz system contains a compiler that generates Oz code (PROGRAMMING SYSTEMS LAB, 1999) from a given protocol specification. A special design focus of the entire system was the clear separation between protocol and application that uses it. Protocols are generic specifications and can be used in many different contexts, the Protoz specification language therefore allows to specify the protocols independently of the particular application.

In the Protoz system, a protocol is given by a collection of roles where each of these roles is specified as an extended finite state machine. Extended finite state machines are normal finite state machines plus (typed) variables. The state in the finite state machine has a set of associated variables that can be queried and/or manipulated in the transition specifications.

The state machine transitions fire upon incoming messages; messages can stem from other agents or from internal procedures. These internal procedures implement the connection to the application and allow for a uniform modeling of internal and external communication. The protocol definition follows the process model shown in Process Model 5.

Process Model 5 Protoz Protocol Design

(1) **Channel definition**
In the channel definition, the roles are declared that participate in the protocol and also the messages that are allowed for each role.

(2) **Role definition**

(2.1) **Declaration part**
The declaration part of a role definition declares the number of instances that are allowed to exist within the protocol, the initial parameters of the role and the states that are used within the extended finite state machine. Furthermore the variables and their types (`boolean`, `integer`, `string`, `message`, `agent` and Lists over the base types) and the names of the application procedures as well as their signatures (parameters, return values) are declared.

(2.2) **Definition part**

(2..2.1) **Transitions**
For each transition, the entry constraints, the actions on entry and exit transitions to other states are defined. An outgoing transition is described by the name of the destination state, the conditions that must be satisfies for the transition to fire and the actions that are executed when the transition fires. Conditions for transitions can be incoming messages or application procedure calls.

(2..2.2) **Actions**
The actions that are executed on the entry to or the exit from a particular state can be *messages* that are sent to other agents or to internal procedures, *Variable manipulation* such as value assignment or various operations on lists over those scalar types or *Branches & Iterations* such as (`if-then-else`) statements for branches and a `while` statement for loops.

(3) **Error handling**

(3.1) **Timeouts**
Timeouts are used to abort the waiting process for incoming messages that do not arrive. Protoz provides two sorts of timeouts:

- *every*: The timeout is triggered on every entry to the state, the timeout is reset on every subsequent entry to the state.
- *once*: The first entry to the respective state triggers the timeout mechanism, the timeout is not reset when the state is entered several times .

The difference between these two classes is in their behavior when a state waits for several messages to come in. For example, when a state waits for five incoming messages with a timeout of 20s, the maximum waiting time for timeout type *every* is 100s because the timeout is reset on every incoming message. The maximum waiting time for timeout type *once*, on the other hand, is 20s because the timeout is not reset on every incoming message.

When a timeout fires, a special message is sent to the waiting state that indicates that the timeout was reached. Subsequent actions are defined in the respective state.

(3.2) **Exceptions**
Exceptions are used to handle errors or failures during protocol execution (eg. unexpected messages, premature protocol termination etc.). Protoz provides a single level exception mechanism that activates an application specific procedure whenever an exception is raised.

5.5.1.3 Interaction Design

We will now provide a characterization scheme of the general nature of the interactions in the application in order to support the system designer in selecting the most appropriate forms of interaction to be used. Interaction can be characterized according to three dimensions.

Purpose The interaction purpose describes why the agents should interact with one another. Thus, the purpose of interaction can either be *co-operation* or *competition*. In the first case, the co-operative behavior can be built into the agents by the designer or it can develop over time in that the agents learn that co-operative behavior usually pays (ROVATSOS AND LIND, 1999) (ROVATSOS, 1999). In the case of a competitive interaction scheme, the agents usually interact to optimize an external property and simulate a market-place where (abstract) prices for resources are negotiated.

Interoperation The agents in the multiagent system either interact *directly*, for example by direct peer-to-peer communication or they interact *indirectly* via the environment. Direct interaction usually requires some facilitator that brings together the agents that want to interact with each other. Such a service is often provided by the agent management framework, eg. FIPA (FIPA, 1997). Indirect interaction, on the other hand, is typical for a real or a virtual environment where the agents meet for some purpose. I is not limited to verbal communication alone; the agents can communicate without explicit message exchange making it more flexible and suitable for agents that have only limited communication skills.

Structure of the problem domain The structure of the problem is mainly characterized by its *coherence*. A very dense problem space is usually *non-separable* ie. it cannot be split into several parts and is thus resistant against task decomposition. A *separable* problem space, on the other hand, often offers a natural decomposition into several independent parts that can be worked on in parallel.

Scalability Requirements An important issue in the interaction design is the scalability of the interaction scheme (LEE ET AL., 1998). In small systems, it is feasible that any agent can communicate with any other agent in the system. But as the systems grows, the exponential growths of message exchange that is introduced into the system can easily become prohibitive. The question of scalability is usually handled in the Society view of the system but it also play are role in the design of the interaction schemes.

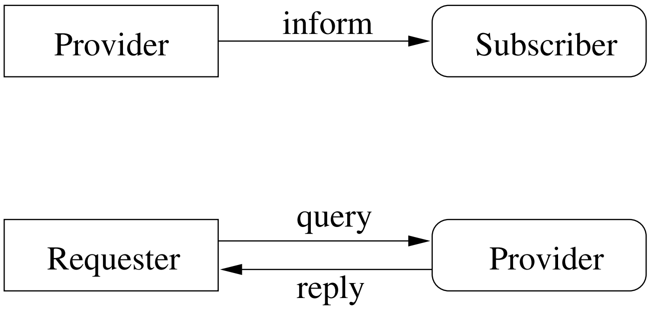


Figure 5.13: Information Exchange

If the system is likely to be large, it is necessary to find scalable forms of interaction in order to ensure later the scalability of the entire system

General forms of interaction have been discussed at various levels of abstraction in the existing literature, a good overview is provided in (KENDALL, 1998A) and, more specifically for workflow and business processes, in (KENDALL, 1998B). We have identified a number of generic interaction schemes that cover most of the variety of specialized forms of interaction. A particular form of interaction that is characterized by the above characterization scheme can then be mapped upon a generic interaction scheme that is subsequently instantiated according to requirements of the target system. The generic interaction schemes that we have identified are discussed in the following paragraphs.

Information Exchange The simplest form of interaction is to exchange some sort of information between the participating parties. As shown in Figure 5.13, this sort of interaction can have one of two forms. In the first form, the information provider sends the information to the subscriber without explicit request from the subscriber. This form of interaction is useful when the subscriber needs to be informed about some sort of changes without knowing in advance when a change will take place. Note that the subscription process that is not shown in Figure 5.13 can be *explicit*, ie. the subscriber asks for being informed about new information or *implicit* in which case the provider broadcasts the information to all reachable receivers regardless of whether they want the information or not.

In the second form of information exchange as depicted in Figure 5.13, the requester sends an explicit query to the information provider specifying in which information it is interested. The information providers answers with a reply that contains the requested information given it is accessible

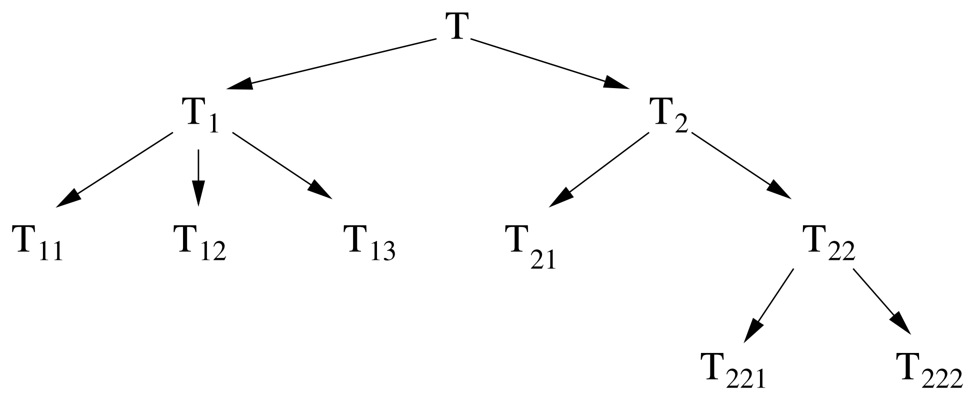


Figure 5.14: Task Decomposition

to the provider and the requester is authorized to receive the information in question.

Task decomposition This interaction scheme achieves a hierarchical decomposition of the overall system task into sub-tasks. Ideally, these sub-tasks are independent from each other and can thus be worked on separately. Figure 5.14 illustrates the basic idea: the overall system task T is split up into two sub-tasks T_1 and T_2 that are divided in turn.

This form of interaction is well suited for separable problem domains. The task decomposition will usually require some sort of explicit co-ordination and thus a certain degree of direct interaction between the agents is desirable but not necessary. The purpose of the interaction can be either competitive or co-operative and the approach scales rather well and can thus be used for large agent systems as well.

Market-based Market-Oriented Programming (WELLMAN, 1996) is the attempt to capture problems of distributed resource allocation with little information. The major advantage of this approach is that the agents need not provide their internal information to external authorities as the only information that is exchanged is the price.

Market-based mechanisms can be implemented in a centralized form as shown in Figure 5.15 where the agents interact with a matchmaker agent that is responsible for bringing together sellers and buyers. This form of Market-Oriented Programming has the advantage that the degree of communication is low because all agents are linked to the matchmaker and furthermore, the trade can be performed anonymously.

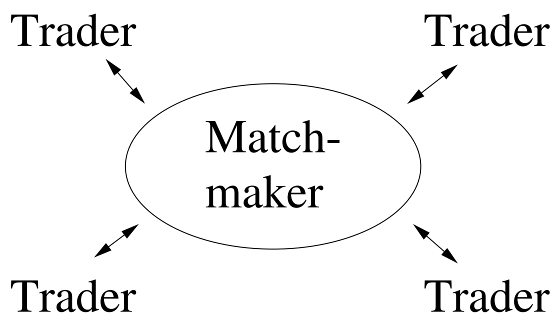


Figure 5.15: Centralized Market

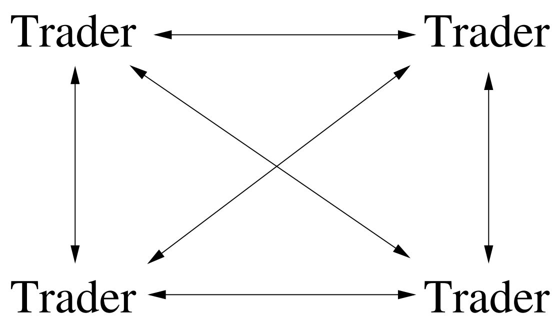


Figure 5.16: Distributed Market

In its decentralized form as shown in Figure 5.16, each agent is linked to any other potential trading partner and the agents negotiate directly without using an intermediate matchmaker. Although this approach has a higher degree of communication overhead it has the advantage that no global pricing scheme takes places which can help the individual agents to increase their profits.

In any case, Market-Oriented approaches are well suited for competitive agent societies that interact directly. The structure of the problem domain is not as relevant but the idea of a market suggests that a separable domain is preferable. Whereas a decentralized market scheme is suitable for small agent societies, a centralized scheme scales rather well and can thus be better used for larger societies.

Blackboards Blackboard architectures (ENGELMORE AND MORGAN, 1988) date back to the early days of distributed problem solving and have undergone

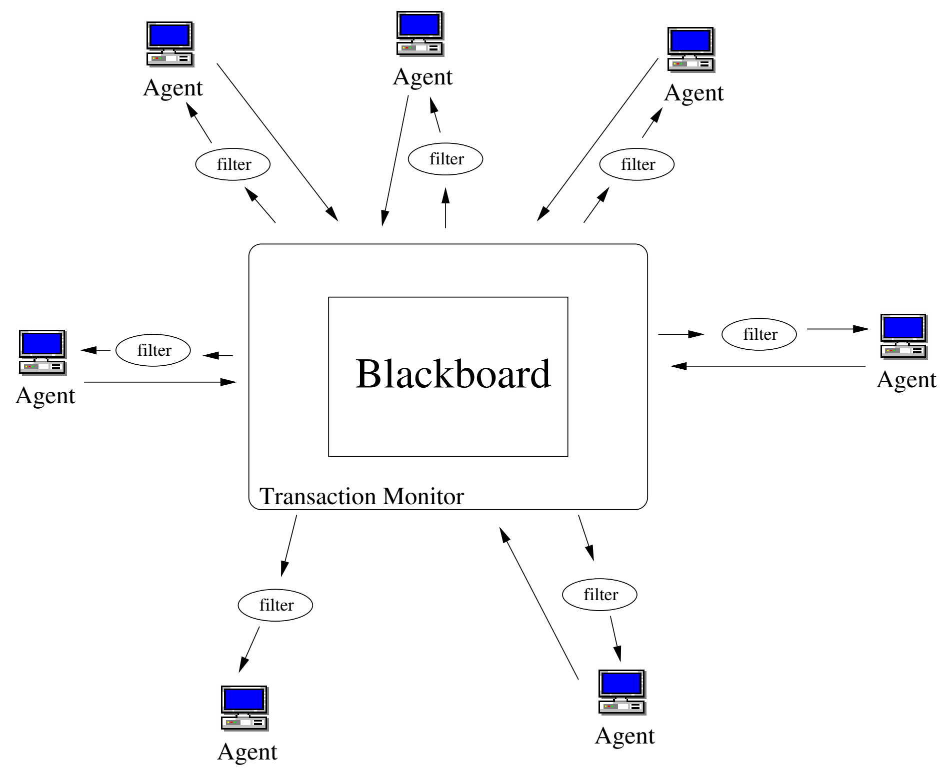


Figure 5.17: Generic Blackboard Architecture

intensive research since then. In its generic form, a blackboard is organized as shown in Figure 5.17. Several agents are connected to a central authority that encapsulates a particular data structure. The transaction monitor shown in the figure is responsible for controlling the access to the joint data structure and to ensure the validity of each access. Each of the agents connected to the blackboard can read and manipulate the joint data. However, the perception (reading) of the data is not necessarily the same for all agents. Each agent receives only data that has undergone a filter process that limits the perception of this agent. This individualized scheme enables the designer for example to use blackboard architectures as the basis for distributed virtual worlds as we will discuss in Section 7.2.1.

Blackboards are suitable for a wide range of problem domains that have a non-separable structure. The approach works for either competitive as well as co-operative settings and is not limited to a particular form of interoperation. Furthermore, blackboard architectures usually scale well and are thus suitable for large agent societies. As we have said above, blackboards can be used to implement virtual words and thus simulate a indirect interoperation scheme.

5.5.2 Case Study

Up to now, the agents in the TCS/MAS system plan only locally, ie. each agent plans for its own goal without notice of the other agents. In this section, we will present an optimization process that is executed by the agents and that reduces their local cost as well as the total cost of the agent society. The key optimization technique in the scenario presented is to find agents that use the same location routes and to have these agents share the respective routes by coupling together at the beginning and by splitting up afterwards. This peer matching is achieved by negotiation processes between the agents in the agent society. In our system, we have combined two negotiation protocols to achieve the global optimization of the schedule.

A schedule is generated in the following way: an initial solution for the module schedule is obtained by running the *contract-net* (SMITH, 1980) protocol whenever a new task is announced to the system. New tasks are incrementally integrated in the existing scheduling which guarantees, that always a solution for the problem (as far as it is known to the system) exists. However, this solution may be (and usually is) not optimal. In order to improve the quality of the existing solution, the *simulated trading* (BACHEM ET AL., 1992) (BACHEM ET AL., 1993) protocol is run on the set of tasks (or the respective modules) currently known to the system. Unfortunately, executing the simulated trading protocol is a computationally expensive operation and so it is executed only periodically — either after a fixed number of new tasks has been added to the existing solution or explicitly triggered by a user request.

The hierarchical organization of the two negotiation protocols supports the aforementioned properties of incrementality and anytime behavior and also implies a high degree of flexibility because of the possibility to asynchronously execute the simulated trading process at any time. In the following sections, we will present the instantiation of contract-net and simulated trading protocol for our specific scenario.

5.5.2.1 Contract-Net

In the context of the contract-net protocol (SMITH, 1980) depicted in Figure 5.18, there are two types of participants: one *manager* and a group of *bidders*. The protocol is initiated by the manager which sends a description of the task under consideration to the bidders. Note, that "task" is not a transportation

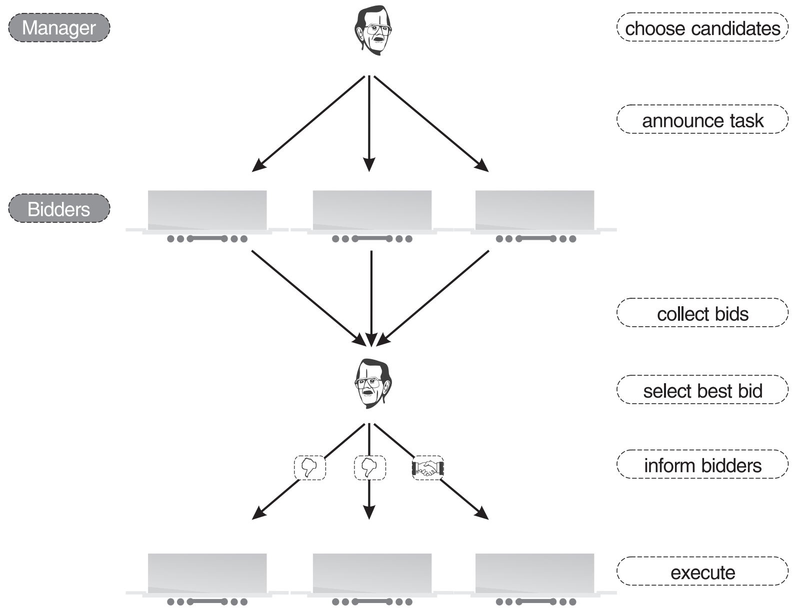


Figure 5.18: Contract-Net Protocol

task mentioned earlier but rather some abstract description of a problem to be solved. We will present the instantiation of the general protocol to our scenario later.

After the bidders have received the task description, each of them computes a bid that informs the manager about costs that will be charged if the task is assigned to that particular bidder. After all bidders have submitted their bids to the manager, the manager selects the bid that minimizes his cost and assigns the task to the respective bidder (+) and rejects the offers of the other bidders (-).

In our system, the contract-net protocol (SMITH, 1980) is executed whenever a new (degenerated) union has computed its local plan as described in Section 5.7.2.2. The union then initiates the contract-net protocol as the manager and offers the plan to the other currently active unions. These unions check if they contain one or more modules that are a potential sharing peers and if this is the case, they offer a sharing commitment to the new union. The new union collects these offers and selects the one that has the largest cost saving potential. It then transfers the module to the winning union and ceases to exist because it does not contain other modules. If no union offers a sharing commitment, the new union remains active as degenerated union.

In order to implement the protocol with the TCS/MAS system, the informal notation given in Figure 5.18 must be formalized to derive the necessary design decisions. In Figure 5.19, the Contract-Net protocol is re-written using the UML. The diagram shows the flow of control and the points of interaction among the participating roles. The joint representation is then split up into two independent finite state machines for each role as shown in Figures 5.20 and 5.21. These finite state machines are then described using the Protoz protocol language and the compiled to directly executable Oz code. The Protoz specification for the Contract-Net is included in Appendix B.

The interaction roles that occur within the Contract-Net protocol are discussed in the following paragraphs.

Bidders A major step in the contract net protocol is on the side of the bidders and it is to identify sharing opportunities with a newly created module. In Section 5.2.2.3 , we have already given a rather detailed explanation of the operator that is capable of finding a maximum degree of overlapping among an arbitrary number of plans. This operator is now used to compute the cost functions of the bidders.

Manager To decide which of the potential contractors $U_1, \ldots, U_k$ is best from the point of view of the manager M, the manager selects the union U_i where $c^-(M, m) - c^+(U_i, m)$ has a maximum.

The contract net protocol presented in this section is used in our system to obtain an initial solution for the scheduling task. However, this solution is usually not optimal and can thus be improved further. This improvement process is done with a simulated trading approach presented in the next section.

5.5.2.2 Simulated Trading

The simulated trading protocol is an algorithm designed to improve existing solutions, not to construct new solutions from scratch. In our case, the input and the output of the protocol are valid schedules where the cost of the output are always less or equal to the cost of the input. This is trivially true since the output can always be the input if no cheaper schedule exists. However, this property is nonetheless important because it guarantees, that the protocol can be aborted at any time and still yield a valid solution. Furthermore, if the protocol is given enough computation time, it is guaranteed to find the optimal solution in the limit. Now, how does this work?

The protocol shown in Figures 5.22 and 5.23 is initiated by a special agent, the *stock manager*. In the course of protocol execution, the agents (here called

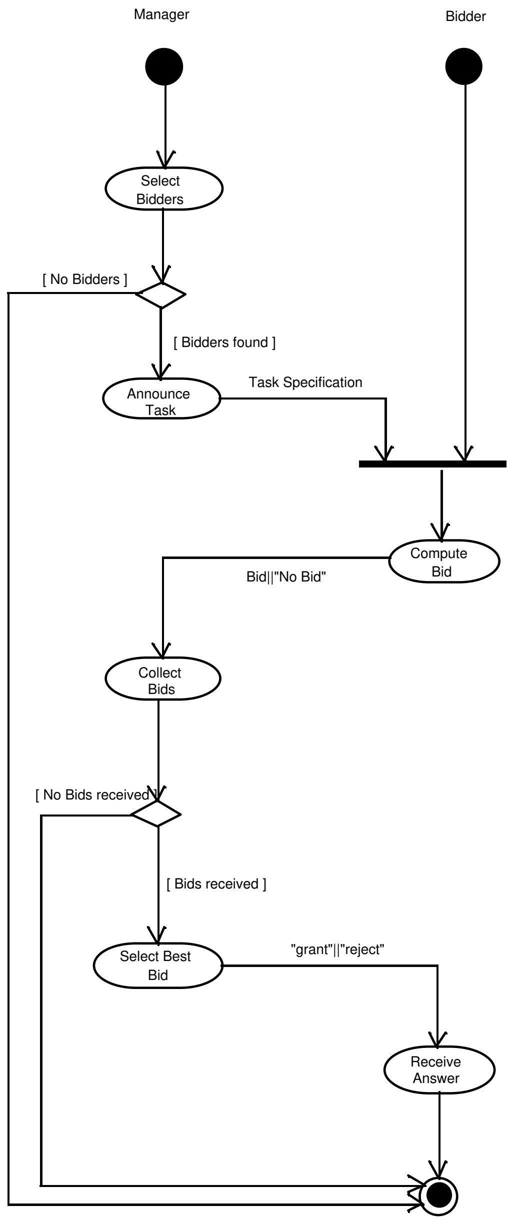


Figure 5.19: Contract-Net Protocol (UML)

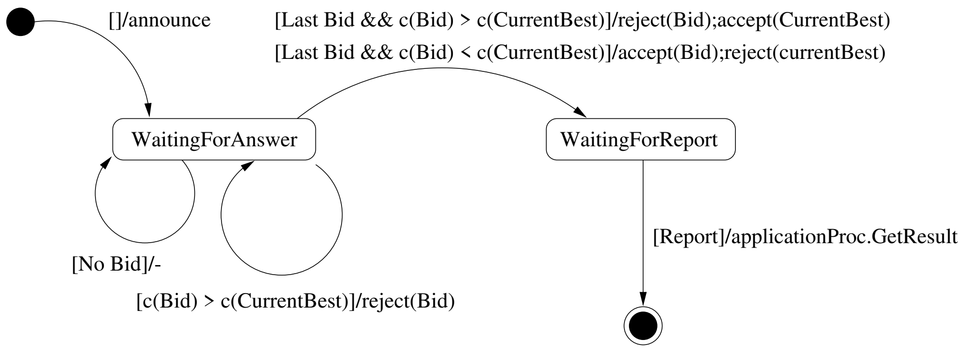


Figure 5.20: Contract-Net Manager

traders) perform several rounds of hypothetical trading, i.e. the traders either choose to sell some of their goods or to buy something from others. In our context, a sell operation corresponds to removing a module from a union and a buy operation corresponds to integrating a module in a union. Thus, the unions try to optimize their cost by exchanging unprofitable modules with better ones. The decision which module to sell depends on a probability distribution induced by the potential cost reduction if the module was sold. Vice versa, the decision to buy a module offered by another union depends on the potential cost reduction if the module would be integrated into the union.

Simulated Trading takes place in several rounds where each union decides whether to sell a module or to buy a module offered by another union. After a fixed number of trading rounds, the stock manager has collected the hypothetical sell and buy actions and it must find a valid *trading match* in the set of actions. There a several validity requirements for a trading match e. g. the difference between deletion and insertion costs must be positive, or there must not be two buy actions on the same sell operation, etc. Finding a trading match is NP-complete (BACHEM ET AL., 1992) and accounts for the computational complexity of the simulated trading protocol. Usually, however, the size of the trading graph still allows to find a trading match within a reasonable time[1]. If a trading match is found, the stock manager informs the traders which actions must be executed, i.e. which modules must be exchanged.

[1]For a 5000 task problem, the execution of the simulated trading protocol with four rounds takes approximately 40 minutes on a Sun UltraSparc.

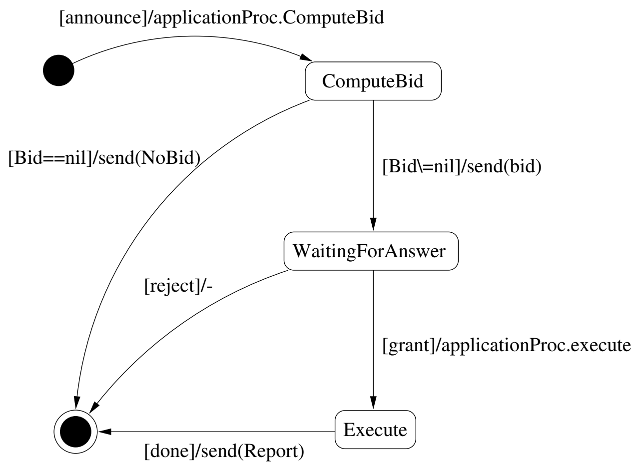


Figure 5.21: Contract-Net Bidder

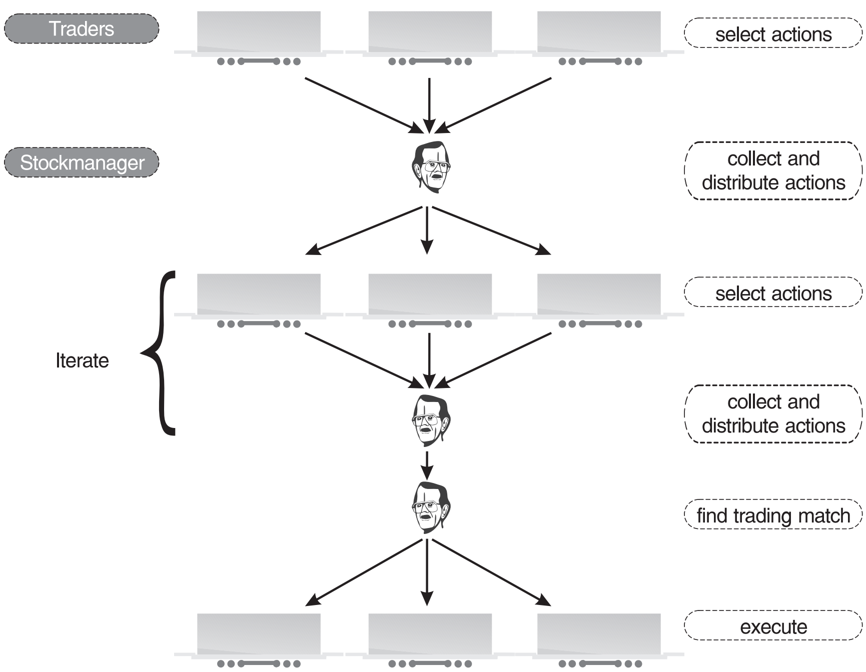


Figure 5.22: Simulated Trading

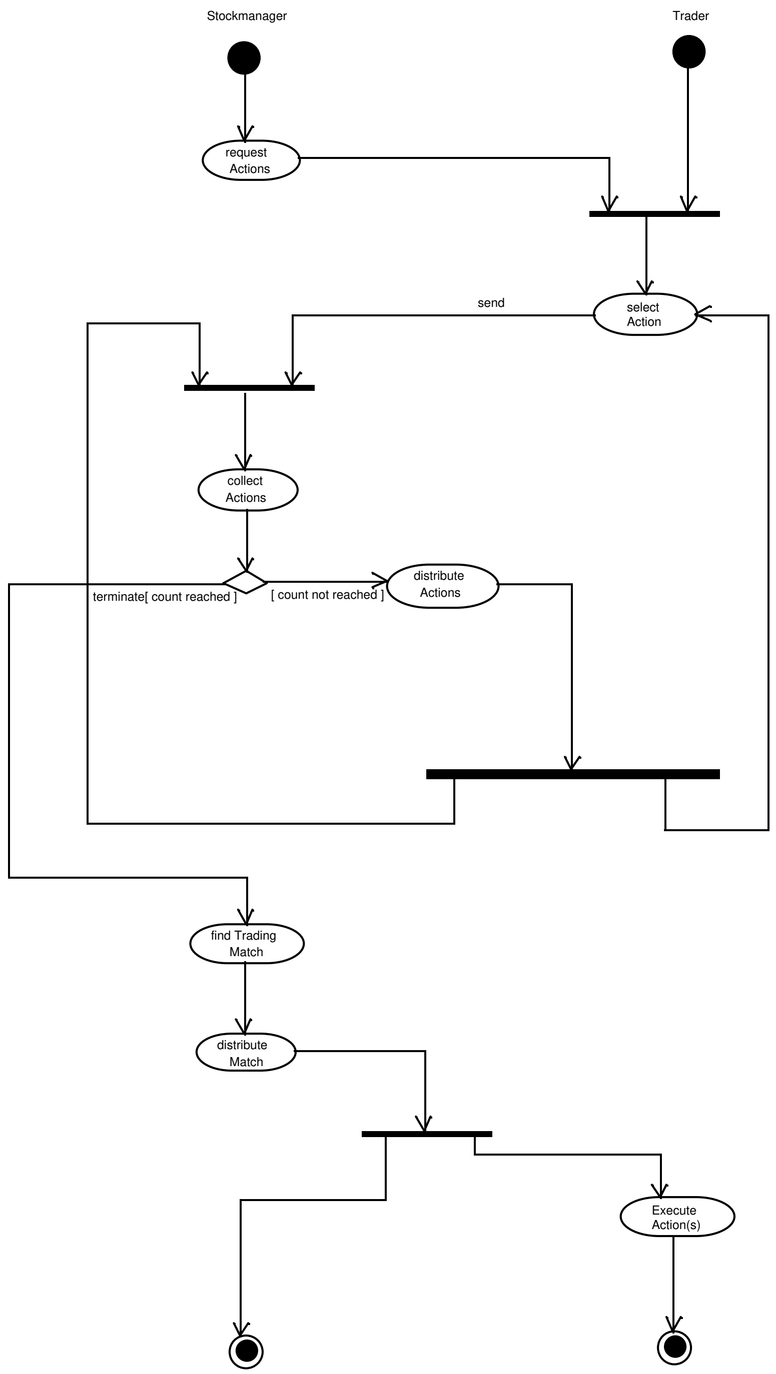


Figure 5.23: Simulated Trading (UML)

5.5.2.3 Client Server

The net manager offers a stateless two-step protocol for the allocation of location routes:

check This message type sends a location route identifier, a time window during which the route should be allocated and the minimal speed by which the route should be traveled to the net manager. Note that the time window is usually larger than the time required by the querying union to travel the location route (the required time is determined by the minimal speed and the length of the route). It is now the task of the net manager to find a time window of the appropriate length that fits in the existing route allocations made by other unions. If it is possible to find such a window, the net manager returns a positive indication to the unions, otherwise a negative indication.

book This message type is used by the union to book a particular location route for a given time window. Note that it does not require a prior **check** operation although it is highly unlikely that a union will find a free time window on a route if it omits the prior check. Note also, that multiple **book** operations on the same location route but by different unions are handled in a first-come first-serve manner. This is due to the fact that the net manager is organized as stateless server, ie. it does not keep track of the order of check operations. The advantages of a stateless organization are that it limits the memory usage of the net manager agent and that is invulnerable to system failures.

free A location route that was allocated by a particular union can be freed by this union if the ongoing planning process makes the use of location route obsolete or when the time window during which the route is needed has changed.

5.6 Society View

5.6.1 General

This view is concerned with the structure of the agent society that implements the target system. In Section 2.2.2 we have already defined the concepts “structure”, “society” and “social system”. In this section, we will provide a characterization scheme for agent societies that allow the software developer to describe an existing society or the specify the requirements for the society of the target system. Furthermore, we will also discuss a micro process model for the construction of an agent society according to a given characterization.

5.6.1.1 Characterization of Social Systems

In the field of multiagent systems, the definitions for agent societies and social systems are much simpler then those in sociology as they were given in Section 2.2.2, but this does not necessarily help the system designer to develop the right view on the agent society. In (WERNER, 1989), the agent society is defined as the (static or dynamic) assignment of roles to agents. 10 years later, in (WEISS, 1999) not very much has changed: "... a specification and assignment of roles and responsibilities to participants in a cooperative planning and/or problem-solving endeavor. ... ". Because of this shortcoming in the multiagent literature, we will try to give a conceptual framework that allows the designer to characterize or model the agent society according to a unified terminology.

An agent society can be classified along four dimensions.

Type The society can be either *open* or *closed*. In an open society, agents can freely enter or leave the society. The entry criteria can be defined by the system designer or they can be defined by the agents themselves. In a closed society, on the other hand, the members of the society are fixed and no fluctuation either into or out of the society takes place.

Structure In a *flat* society, the interaction between agents is unlimited, ie. each agent can interact with any other agent. While this structure provides a maximum degree of flexibility, it also implies a high degree of overhead. For example, in an agent society that uses some sort of explicit communication, a flat society structure requires $n(n-1)$ communication channels.

A *hierarchical* society is not as flexible as a flat society because the interaction channels between the agents are limited. Usually, some agents have special organizational roles and are responsible to filter and/or limit interactions. In Figure 5.24, an example for a hierarchical society in the RoboCup (NODA, 1995) domain is shown. Within each sub-structure (defense line, middle field, offense line), the agents can freely communicate with any other agent of that sub-structure. Each sub-structure also has an agent with the organizational role of a "local captain" (indicated by the single boxed) that is responsible for the communication with the team captain (double box) who in turn manages the communication with the coach. The goalie has a special position in that he can communicate with the captain directly.

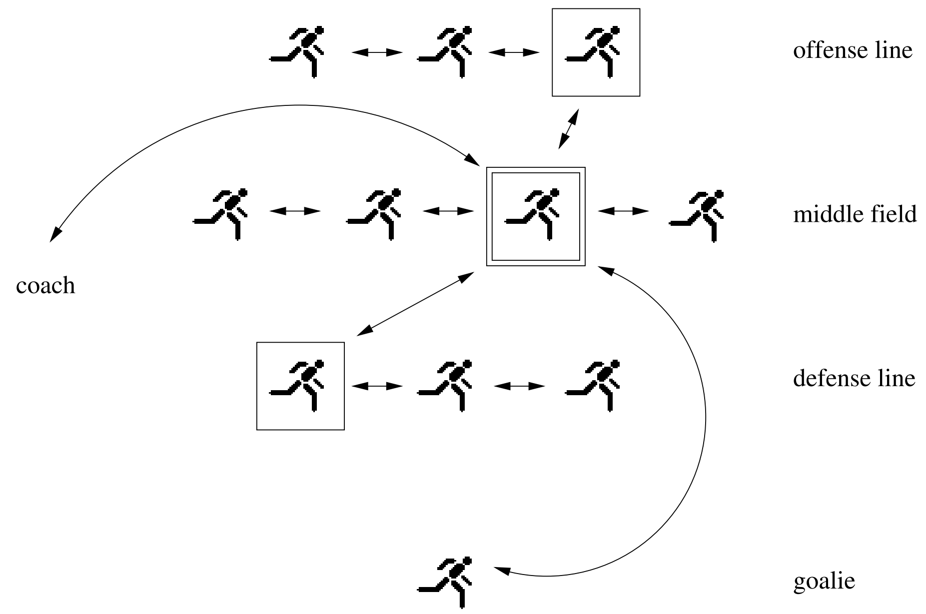


Figure 5.24: Example for a hierarchical society

Instead of 132 connections in the case of a flat society, the resulting structure has only 11 connections and thus less overhead. The drawback of this solution however, is that in the worst case, messages from an individual player to the coach and vice versa must be forwarded through three intermediate agents.

Consistency This property refers to the agents within the society and differentiates between *homogeneous* and *heterogeneous* societies. In a homogeneous society, all agents have the same external properties such as the underlying architecture or the set of behaviors that they have at their disposal. Even more important is the fact that the agents all have the same knowledge representation which enables the agents to freely exchange messages without complex and error prone translation mechanisms.

In a heterogeneous agent society, the agents can have different architectures, behaviors and knowledge representation. Especially the latter makes the interoperation between agents with different architectures difficult.

Temporal context The temporal context defines how the structure of the agent society develops during its lifetime. In a *static* society, the structure is fixed from the beginning and does not change during the operation of the system.

In a *dynamic* society, the structure and the interconnections between the agents can change either because of a pre-defined dynamic behavior specified by the system designer or due to a self-adaption process of society in order to achieve a higher performance. Self-adapting agent societies are a promising technology in various application fields (LIND ET AL., 1999C),(GERBER ET AL., 1999B) and especially *holonic* approaches (GERBER ET AL., 1999A) become increasingly interesting.

5.6.1.2 Designing Social Systems

We will now give a prescriptive process model on how to develop the optimal social structure for the target system. Firstly, however, we must clarify what the term "optimal" means in this respect as no single organizational design is optimal for all purposes (RAWSON, 1992). According to (CARLEY, 1999), optimality of a social system depends on:

- The *tasks* to be performed,
- the *agent capabilities*,
- the *characteristics of the environment*,
- *external constraints* (eg. legal, political) and
- the *goal* of the optimization process, eg.
  - efficiency
  - effectiveness
  - accuracy
  - minimal cost

The first four aspects in the above list are dealt with in other views within the MASSIVE method leaving the *goal* aspects to be handled in this view. According to Figure 5.25 the performance measure of an agent society depends on its structure and the behavior as well as on the requirements that imposed by the environment and quantified in the attributes described above. Thus, the optimal society structure is expressed in terms of the above goal characterizations eg. by specifying maximal communication bandwidth, channel capacity, number of available channels, quality of services or message costs.

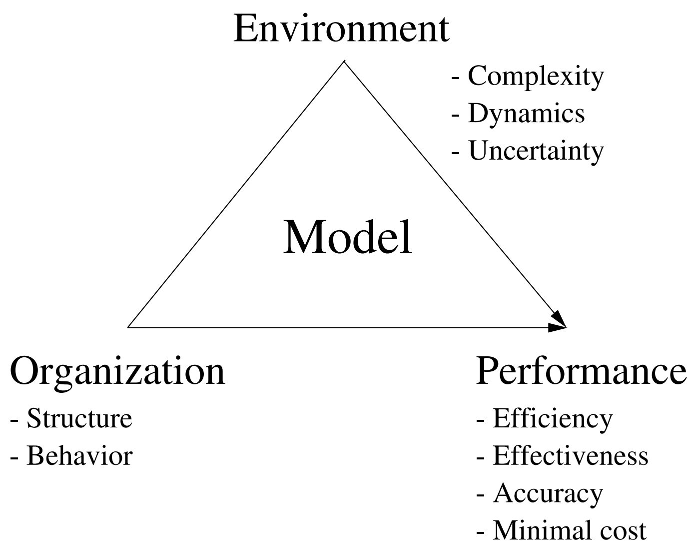


Figure 5.25: Social System Performance (adapted from (SO AND DURFEE, 1998))

The social structure is then constructed according to the six step process shown in Process Model 6

Although Process model 6 describes a very general way for obtaining the optimal society structure, it does not describe the actual means the designer has in order to impose a particular structure onto the system. This aspect is not covered in this section as this is subject of a research field named *Computational Organizational Theory* (CARLEY, 1999) to which we direct the reader for the solution of a particular problem. In the next section, we will present the society design for the TCS/MAS system where will also discuss a popular means for decomposing large agent societies.

5.6.2 Case Study

According to the general classification scheme presented in the previous section, the society within the TCS/MAS system is

- *homogeneous* because all agents share the same architecture and the same knowledge formats,
- *semi-open* because newly created agents enter the agent society but no agent ever leaves the society,

Process Model 6 MASSIVE Society Design

(1) **Identify**
In the first step, the goals of the optimization process are identified according to the above characterization.

(2) **Quantify**
The goals that have been identified in the prior step are now quantified in order to enable the decision on whether the optimization has led to satisfactory results or not.

(3) **Construct**
Now, the a social structure is set up that is believed to satisfy the goals at the required quality level.

(4) **Implement**
The social structure that was developed in the previous step is implemented.

(5) **Evaluate**
The structure is evaluated with respect to the quality requirements.

(6) **Iterate**
If the social structure fulfills the quality requirements, the process is aborted. If the results are below the previously defined quality standard, the process is iterated from step 3.

- *dynamic* because the structure of the society changes according to the holonic union formation process, and finally,
- *flat* because any agent can negotiate with any other.

Especially the last point needs some further discussion as the society structure of the TCS/MAS system was only initially flat until we discovered, that this design lead to serious performance losses in larger societies. The intended system of the system are a maximum of 5000 agents and it is easy to imagine what communication overhead results from a flat society structure in a society of this size.

In the case of a new task that is announced to the system, any union that is currently active will be contacted by the Contract-Net manager in order to to check whether it has sharing opportunities with the new task or not. Clearly, it is a better idea to limit the class of potential bidders to those who have a potential chance to become a sharing peer. This selection process that is executed whenever a Contract-Net is initiated is called *dynamic* clustering.

During the optimization phase of the existing schedule that uses the Simulated Trading approach, the situation is even worse because Simulated Trading is computationally expensive because the individual unions must execute a lot of plan integration operations in order to decide whether they will want to buy some offered module or not. In this case, it is important to partition the society into groups of agents that can become potential trading partners in order to limit the number of checks that must be performed by a single unit. This partitioning process that is executed before a trading phase is called *static* clustering.

In the following sections, we will explain the algorithms for these two clustering mechanisms in detail.

5.6.2.1 Dynamic Clustering

The goal of the dynamic clustering algorithm is to select the potential partners for a new degenerated union that wants to find sharing partners.

DEFINITION 5.6.1 (MAXIMUM TIME WINDOW) *The* **maximum time window** *of a union $U_j^{i_0,\dots,i_n}$ is given the lowest lower and the highest upper bound of any module(s) in the union. More formally, let $\underline{t}_i = M_i.TW.\underline{t} \forall i \in \{i_0, \dots, i_n\}$ and $\overline{t}_j = M_j.TW.\overline{t} \forall j \in \{i_0, \dots, i_n\}$. Then $TW(U_j^{i_0,\dots,i_n}) = [\min_{\underline{t}_i}; \max_{\overline{t}_j}]$.*

DEFINITION 5.6.2 (NODELIST) *The nodes the occur within a plan P of a module M are obtained by applying the $Nodes$ function to the plan, ie. $Nodes(M.P) = \{N_1, \dots, N_n\}$.*

The Nodes of a union $U_j^{i_0,\dots,i_n}$ then compute to
$Nodes(U_j^{i_0,\dots,i_n}) = \bigcup_{i \in \{i_0,\dots,i_n\}} Nodes(M_i.P)$

Using these definitions, we can specify the dynamic clustering algorithm for a new union U as shown in Algorithm 1.

Algorithm 1 Dynamic Clustering

```
P := 0
for all U' ∈ 𝒰 do
    if TW(U) ∩ TW(U') ∧ Nodes(U) ∩ Nodes(U') ≠ ∅ then
        P := P ∪ {U'}
    end if
end for
```

5.6.2.2 Static Clustering

The idea of the static clustering algorithm is to partition the agent society into several groups that share particular features. The higher the relation between the individuals in a cluster and the higher the differences between the clusters are, the better is the clustering. The idea of clustering is illustrated in Figure 5.26 where we can see twelve unions that are arranged in three clusters. The

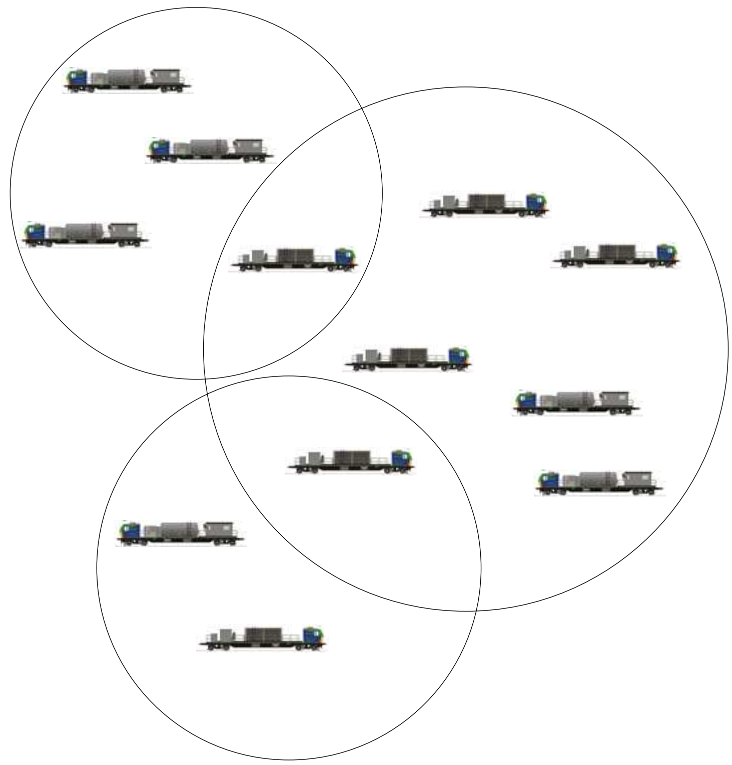


Figure 5.26: Clustering

clustering is not unique because two unions can be assigned to two different clusters each.

However, to find an optimal clustering is an NP-complete problem and thus only feasible in small societies. In the TCS/MAS system with up to 5000 agents, we have to apply heuristics to find a rather good, though usually not optimal clustering. The idea of the clustering algorithm shown in Algorithm 2 is to pick arbitrary unions from the set of all unions as long the newly picked unions have no time window or location route (ie. common nodes) overlapping with previously picked unions. These unions become the core elements of the clusters. If no overlapping-free new union can be found, the remaining unions are assigned to the cores according to some distance measure d.

To improve this algorithm further and to make it more robust, we have implemented some additional features. One of these improvements addresses the fact that the algorithm as presented in Algorithm 2 does not clearly specify how the initial core elements are chosen. The wrong choice however, can endanger the success of the entire clustering idea. If one of the first picks of the algorithm

Algorithm 2 Static Clustering

$C := 0$
while $\exists U \in \mathcal{U} : \nexists U' \in C : TW(U) \cap TW(U') \neq \emptyset \wedge Nodes(U) \cap Nodes(U') \neq \emptyset$ **do**
  $C := C \cup \{\{U\}\}$
  $\mathcal{U} := \mathcal{U} - \{U\}$
end while
while $\mathcal{U} \neq \emptyset$ **do**
  pick $U \in \mathcal{U}$
  find $\{U_{i_0}, \ldots U_{i_n}\} \in C$ such that

$$\forall \{U'_{j_0}, \ldots U'_{j_m}\} \in C : \sum_{i=\{i_0,\ldots,i_n\}} d(U, U_i) \leq \sum_{j=\{j_0,\ldots,j_m\}} d(U, U'_j)$$

  $C := C - \{\{U_{i_0}, \ldots U_{i_n}\}\} \cap \{\{U_{i_0}, \ldots U_{i_n}, U\}\}$
  $\mathcal{U} := \mathcal{U} - \{U\}$
end while

is a large union, ie. with many modules, that has a low degree of overlapping, ie. the union has many location routes and thus many nodes, the algorithm will not find many other unions that does not overlap with this large union and consequently, the number of core unions will be small leading to large clusters. To circumvent this difficulty, we have modified the above algorithm in that it prefers small unions over larger ones in the first cycle. The other improvements are only of minor interest and are therefore not discussed here.

5.7 Architectural View

5.7.1 General

The goal of this view is to transform the feature specification of the other views into a system architecture. But what is a system architecture? Because of the fact that there is no widely agreed definition for the term in the Software Engineering community (SHAW AND GARLAN, 1996), (SAUNDERS ET AL., 1996), we have to select one of the existing definitions that is most suitable for our purposes.

DEFINITION 5.7.1 (SOFTWARE ARCHITECTURE) *The* **software architecture** *describes fundamental structural attributes of a software system.*

Although this definition is rather general and leaves some space for interpretation it is nonetheless expressive enough to clearly define the focus of this view which is to model the *components* of the target system and the *connections* between them. The software architecture is therefore a more static description of the code of the target system that must, however, account for the dynamic aspects that were defined in some of the other views. Before we start with concrete considerations of how to find the most appropriate system architecture, we will briefly introduce some general properties that should be found in any architecture that is developed.

Two of the most fundamental metrics that have been introduced several years ago are *coupling* and *cohesion.* According to (CONTE ET AL., 1996), coupling measures the number of interconnections among the entities of the system and cohesion measures the internal relationships of these entities. In a good system architecture, the coupling should be low in order to achieve a high degree of modularization and the cohesion of the individual components should be high in order to keep effects of particular operations as local as possible. Furthermore, a good system architecture should try to distribute the complexity of the target system equally over the entire architecture (ROMBACH, 1994A) as this reduces the complexity of individual components and thus supports the designer's understanding of these components.

Obeying these basic requirements will eventually lead to a system architecture that supports changes because of the reduced complexity of the individual components as well as well as reuse because of the modularization of the architecture.

In the rest of this section, we will address some issues that are characteristic for multiagent systems. Due to the wide range of possible implementations, however, we will not try to suggest a reference architecture (SHAW, 1995). The requirements for a particular system can be so specific and individual that it would be highly unnatural to try to fit it into a standard design. However, we strongly believe that some architectural patterns for specific classes of multiagent systems will evolve during the next years when multiagent technology will become more common in industrial systems.

5.7.1.1 The Architectural Feature Space

In the Architectural view, we are concerned with the problem of how to transform the more or less abstract feature specifications of the other views into a coherent system design. Therefore, this view will have to work on a lot of features that are shared with other views and we will therefore first of all characterize the *design space* of the target system. The design space is defined as a three-dimensional space that arranges the features that are addressed within the architectural view according to three different dimensions.

The first dimension to describe a particular feature is by classifying it according to its *temporal behavior*. Thus, a feature can either be *static* in that it accounts for a structural aspect of the system, or it can be *dynamic* and capture some of the relationships between the static object. Features of the first class will therefore usually relate to components and features of the second class will model the nature of interconnections between components.

The second dimension describes the *level of granularity* of a feature. This can either be the *micro* level that captures properties of individual components or it can describe *macro* aspects, ie. aspects that influence the architecture of the entire system.

The third dimension, finally, differentiates features according to the *purpose*. The purpose of a feature can be *task specific* or it can be *task independent*. In the first case, the feature will typically represent entities that are characteristic for the domain while in the second case, these features can be found in several domains at the same time.

In the following, we have listed some features that are addressed in this view and classified them according to the three dimensional scheme. Figure 5.27 shows the relations between the listed features.

The Architectural view is mainly concerned with the following topics:

Entities No all entities that are part of the system architecture are necessarily agents although the agents play the major role in the target system. However, it is not always a good idea to encapsulate all relevant entities into agents eg. when such an entity is frequently used by several agents, it is possibly a better idea to model it as an object that is accessed by the agents because of the communication overhead that would accompany the approach to model the entity as an agent. Therefore, it is the goal of this view to identify and to model the agents and the other major objects within the target system.

Control flow Whereas the control flow at the agent level is defined by the interaction schemes given in the Interaction view, the control flow at the object level must be coordinated explicitly as well. Because not all entities that occur in the target system are agents, the integration of these entities into the system are modeled in this view.

Information flow The arguments that hold in the case of the control flow can be applied to the information flow as well. Again, the information flow at the agent level is defined in the Interaction view but the information flow between other entities must be modeled as well.

Agent management Agent Management captures all tasks that are concerned with the operational framework of the agents. This framework includes matters such as the agent identification scheme, available service information or firewall technology. Standards such as described in (FIPA, 1998) can be a valuable help for the designer in this respect.

Communication model The low level communication model of the application must be described and integrated into the system. The range of possible communication techniques for a particular application is very broad and the designer must decide on the need for synchronous or asynchronous messages, one-to-one, multicast or broadcast messages etc. The need for particular communication services can have a huge impact on the entire system as we will see in Chapter 6.

Agent architecture This aspect is one of the most important and also one of the most difficult aspects to deal with. Therefore, we have dedicated Section 5.7.1.3 to the detailed discussion of this topic.

Database design Especially larger systems or systems that are supposed to operate in an industrial context usually need storage facilities and mechanisms that cannot or should not be proprietary. Therefore, the designer must use standard database technology to handle these matters. However, although common database systems provide the necessary software support, it is still up to the designer to develop a database structure that suits the particular needs of the target system. Since database design is a long-known topic in the computer science community, we direct the reader to the widely available literature..

External components/devices In order to provide the required functionality to the user, a software system usually relies on external services that are provided either by software or hardware components. It is one of the tasks of the Architectural view to identify these components or devices and to fit them into the overall system design. This integration is achieved by defining the appropriate interfaces and the connections of the external entities to particular entities of the software system.

Each of these features can be characterized according to the three dimensions presented earlier and arranged in a three dimensional space spawned by these dimensions as shown in Figure 5.27.

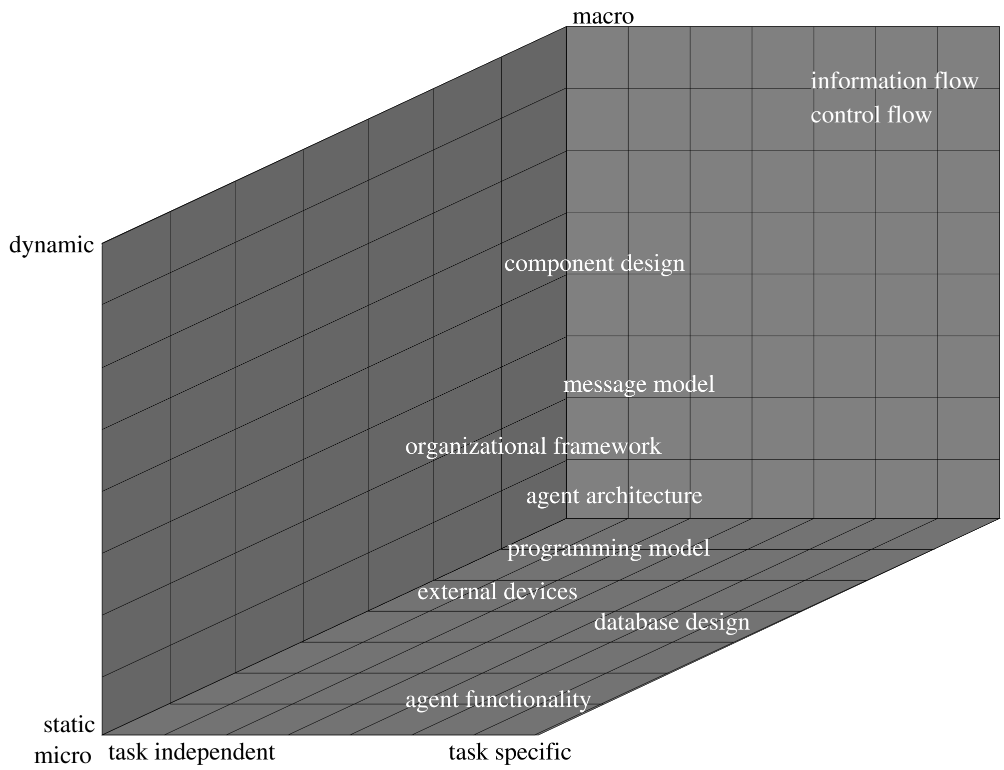


Figure 5.27: Architectural Design Space

5.7.1.2 System Architecture

In order to find the most appropriate architecture for the target system, it is useful to think first about the overall nature of the system. Then, an appropriate *design idiom*, ie. a fundamental pattern that is characteristic for the entire design (SHAW, 1995) is chosen according to this classification.

Several of the most common design patterns shown in Figure 5.28 are discussed in (GARLAN AND SHAW, 1993), (GARLAN AND SHAW, 1994):

Main Program & Subroutines This is probably one of the oldest design patterns, stemming from the time where procedural programming languages were the only available higher level languages on the market. The characteristic feature of this pattern is the focus on the main program that provides the main control loop and acts as a driver for the subroutines attached to it. This type of architecture is still often found, especially in systems with a simple control flow, for example for scientific calculations.

Batch Sequential This architectural pattern is almost as old as the previous

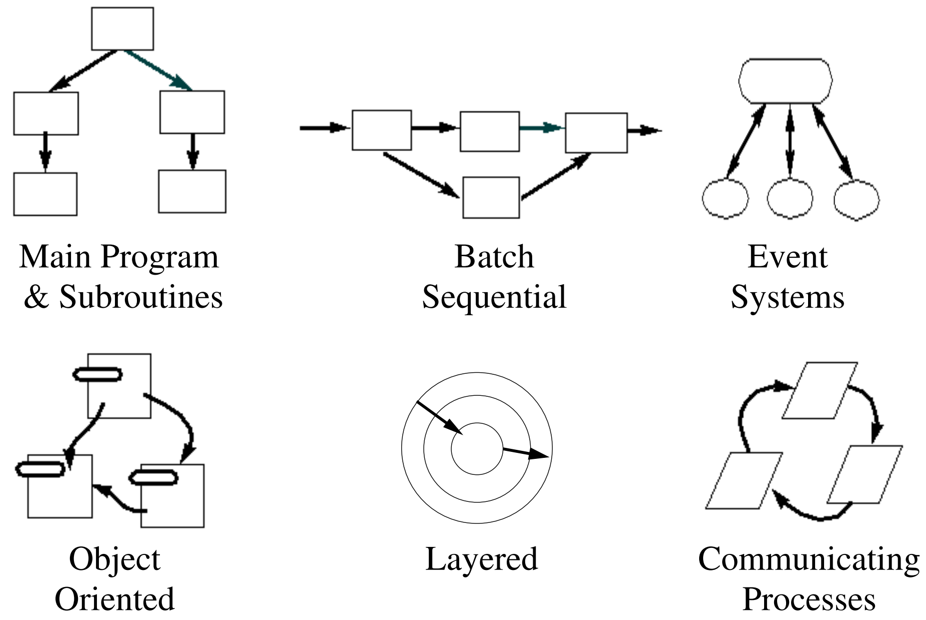


Figure 5.28: Generic Software Architectures

one and has its origin in the time when memory was a valuable good and multi-process operating systems had not yet been invented. In this pattern, each component has a set of inputs and outputs and performs a local transformation process that maps the inputs to the outputs. The components are independent of each other and can be invoked sequentially on a single process machine.

However, this architectural pattern may gain some new support because it is well suited for workflow systems (ODGERS ET AL., 1999) and supply chain management.

Event Systems Event Systems are characterized by an asynchronous, implicit method invocation mechanism that is used by the components to communicate with each other. The idea is that each component registers with an event dispatching system in order to receive particular events that are issued by other components. This mechanism is called implicit method invocation because the component that issues an event does not explicitly request a service by another component, it may even be the case the issuing component does not know which other components react to the event.

The canonical example for event-driven systems are window systems that contain a large number of and require a flexible, localized control flow.

Object Oriented The fundamental abstraction in object oriented systems is the encapsulation of data representations and the associated operations in a single structural entity – the object. Well defined object oriented systems are often easier to understand then systems with a traditional architecture because of the semantic relation of the entities in the system with the entities they are supposed to represent. However, object orientation can easily lead to an over-modularization of the system with fatal consequences to maintenance and runtime behavior. Object-oriented architectures are very common in applications that provide a graphical user interface.

Layered In a system with this sort of architecture, several layers are built on top of each other and a each layer uses services from the lower layer and provides some service to the upper layer. Layered architectures are the first choice when the complexity of the task can be hierarchically decomposed. Furthermore, layered architectures support local changes because of the well defined interfaces between the layers. The most prominent examples for layered architectures are the OSI reference model (THE INTERNATIONAL ORGANIZATION FOR STANDARDIZATION, 1998) for protocol system or the TCP (DEFENSE ADVANCED RESEARCH PROJECTS AGENCY, 1981) architecture.

Communicating Processes This design pattern represents a flexible architecture for systems with several independent, interacting entities. These entities can either reside on the same computer or they can be spatially distributed and connected via a communication network. Examples for this architecture include Client-Server systems as well as multiagent systems.

This list of architectural prototypes is not exhaustive but it represents the majority of the common system architectures. However, it is often the case that a particular system is not implemented according to a single idiom, but according to several of them. For example, a multiagent system can be implemented as communicating processes and also have an object oriented architecture at the same time. In fact, we will see in the case study, that this is the case in the TCS/MAS system.

In some cases, the decision on which of these idioms to apply for a specific target system might be straightforward. In other cases, where the decision is not so obvious, we can only provide as an advice to the system designer to briefly apply the above idioms to the target system in order to develop a feeling for how a system might look that is implemented according to a particular idiom and then to select the most appropriate. Unfortunately, up to now no generally applicable set of guidelines exists that can help the designer in choosing the best architectural idiom.

5.7.1.3 Agent Architecture

On of the most important aspects of multiagent system development is to define the basic entities within the system – the agents. As we have already discussed in Section 5.4.1.2, an agent is a conceptual abstraction the consists of a set of roles and an architecture that implements these roles. Thus, we define the term "agent architecture" as follows.

DEFINITION 5.7.2 (AGENT ARCHITECTURE) *An* **agent architecture** *is a structural model of the components that constitute an agent as well as the interconnections of these components together with a computational model that implements the basic capabilities of the agent.*

To select the the best agent architecture for a given set of roles is as least as a big a problem as to find the best architectural idiom as discussed in the previous section (MÜLLER, 1998). Therefore, we recommend a two step process to identify the best architecture from the numerous architectures that are currently on the market (MÜLLER, 1996A). First of all, the system designer should *characterize* the requirements for the architecture according to fixed scheme that covers the most important issues to be addressed. Secondly, the designer should *evaluate* existing architectures according to the requirements defined in the first step and then select the best matching architecture. We strongly recommend to check out existing approaches before trying to define a proprietary architecture. Only if no matching architecture is found, the burden of designing an implementing a new architecture is justified. Even in this worst case, the developer should consult the existing literature (eg. (JUNG, 1999)) in order to avoid sub-optimal results.

To address the first aspect of this two-step process, we have defined the following set of properties that can be used to characterize the requirements for the agent architecture.

Reasoning capabilities The reasoning capabilities of the agents define the most important property and often determine the overall complexity of the agent architecture. For example, an agent may be forced to *plan* its actions if it is not a purely *reactive* agent, or it may use some *utilitarian* reasoning mechanisms to chose among several possible actions. Another important issue is the ability for an agent to *learn* from past experiences or the agent may be used to fulfill *special tasks* such as theorem proving etc.

Resource limitations This aspect of the characterization describes the resources that are available to a single agent within the multiagent system. If an agent has a limited amount of processor time or memory space, it is impossible to use an agent architecture that requires, say, the resources of a Unix process. However, this point can also be viewed from a different angle. If the individual agent has to deal with very complex problems, a simple architecture may not be able to cope with the resource requirements of the architecture because it was not designed for heavy weight problems.

Control flow The aim of this requirement is to characterize the control flow that is needed within the agent. First of all, the designer should decide whether a *sequential* flow of control is sufficient or if the agent is required to do several things at the same time and thus needs some *parallel* action execution model. In the second case, a concurrent architecture that is in most case much more complex then a sequential architecture must be chosen.

Secondly, the designer must decide about the required *flexibility* of the control flow. In more static setting, the flow of control can be explicitly hard-coded into the architecture while in a dynamic context, the flow of control is likely to undergo changes and must therefore be described implicitly eg. in plan scripts that are interpreted at run time and that can be changed during the agent is in operation if it is necessary.

Knowledge handling The knowledge representation within the target system is defined in the Task view. In the Architectural view, the knowledge structures that are defined there must be characterized in order to decide which architectural features are necessary to effectively handle these structures.

Firstly, it is important if the agents knowledge is stored *explicitly* in a knowledge base or is it encoded *implicitly* into the agent code. Secondly, the knowledge structures may be represented in a *symbolic* manner using some sort of logical formulae or in a *sub-symbolic* form eg. in its simplest form as collection of values or more elaborate in the form of a neural network.

Autonomy The degree of autonomy that is required by the agent defines how the agent interacts with its environment. A *reactive* agent simply responds to external stimuli by reproducing a pre-defined behavior when a particular stimulus is given by the environment. A *pro-active* agent, on the other hand, can become active without external trigger and it then performs some action that satisfies the goal. Pro-active agents are usually more complex and their behavior is not always predictable.

User interaction The more interaction the agent has with the system user, the more elaborate the user interface has to be in order to provide convenient means for input and output data. Furthermore, an advanced user interface agent will perform user profiling and try to learn the users preferences from his or her input/output behavior.

Temporal context This aspect characterizes the agents lifetime. Obviously, an agent with only a limited activation time will need another form of persistence mechanism – if at all – then a long-running agent. Persistence refers to the ability of the agent to maintain knowledge structures over time and over unavoidable down-times due to service failures such as hardware or software crashes. However, the amount of information that is collected by the agent over its lifetime can become very large and must be handled in an effective manner and means must be provided by the architecture to manage the data handling process.

Decision making This attribute characterizes the way in which the agent comes to its decisions during its reasoning processes. While some authors claim that *rationality* is an inherent property of any agent (RUSSELL AND WEFALD, 1991), (RUSSELL AND NORVIG, 1995), there are others who consider architectures that support *emotional* decision making as an alternative (BURT, 1998). There are two main fields for a potential application of emotional architectures. Firstly, they can become valuable tools to implement lifelike characters and avatars that represent a human user in networked environments. Secondly, the notion of emotions can be used to express complex heuristics for advanced software agent in a natural way. However, the development of the basic technology is still in its beginnings and does not play a relevant role until now. Still, the developer of a particular application may want to consider these ideas if they are appropriate for the problem in question.

In this section, we have outlined the basic objectives of the Architectural view with some emphasis on the overall architectural paradigm underlying a particular application and on the agent architecture because of its complexity and because the agents are the main entities in a multiagent system. In the next section, we will demonstrate how these more theoretic considerations are applied in a realistic problem.

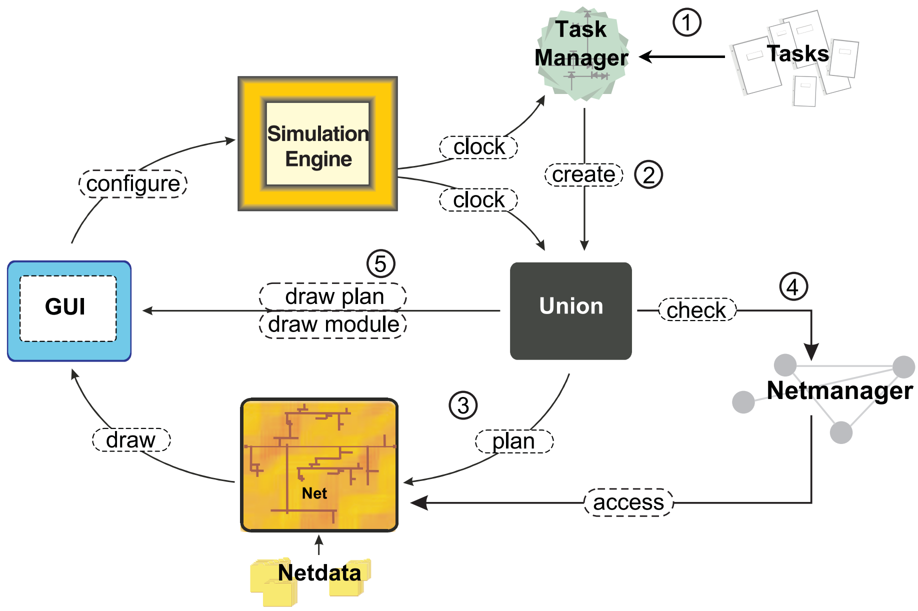


Figure 5.29: System Architecture

5.7.2 Case Study

5.7.2.1 System Architecture

In this section, we give an overview of the architecture of the TCS/MAS system and develop the structure of the target system according to the classification scheme that was introduced in Section 5.7.1.

Entities The most important entities of the TCS/MAS system and their interconnections are depicted in Figure 5.29 and Figure 5.30.

The agents within the TCS/MAS system are created according to the role deliminations that were discussed in Section 5.4.2. The *unions* agents are the main elements of system as they perform the local planning of transportation routes and the local optimization procedures as discussed in Section 5.5.2. The *task manager* is responsible for taking incoming transportation requests and forwarding the to a union that serves the respective task. The *net manager* agent, finally, is responsible or controlling the location route allocation process of the unions.

The TCS/MAS system, however, does not only contain agents that accomplish the system functionality, several important activities are encapsulated into objects for the sake of simplicity and efficiency.

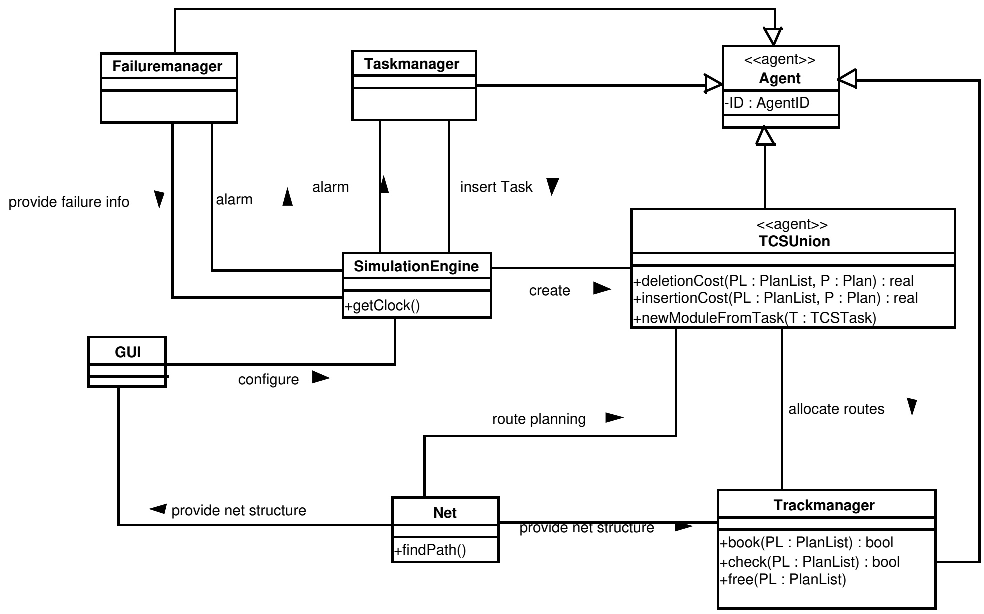


Figure 5.30: System Architecture (UML)

The *simulation engine* object is the main coordination authority of the TCS/MAS system and provides the system wide clock that is needed to synchronize the concurrently active agents and it implements an alarm mechanism that allows the agents to suspend their execution until a particular clock cycle. Finally, the simulation engine supports system development and debugging by allowing the designer to run the system in single-step mode.

The *net* object provides a route planning mechanism that can be used by the agents to find an optimal route between any two nodes in the network.

The *graphical user interface (GUI)*, finally, is responsible for the visualization of the system activities and for the configuration of the system behavior. The visualization of the system state is divided in a static component that allows the user to inspect the current status of the agents (e.g. their current plans etc.) and a dynamic component that allows the user to simulate the execution of the agents plans. Additionally, the GUI provides various configuration options that can be used to change the system behavior.

Now that we have introduced the main objects of our system, we will give a brief description of the data and control flow that is indicated by the arrows in Figure 5.29.

Control and Information flow The numbers shown in Figure 5.29 illustrate the basic cycle that is executed whenever a transportation task is announced to the system. We have left out the plan optimization steps that were discussed in the Interaction view in Section 5.5.2.

The basic execution cycle is the following:

(1) The task manager receives a task description.

(2) The task manager creates a new agent to handle the task.

(3) The new agent requests the optimal path from the net object.

(4) The new agent checks the validity of the resulting plan by querying the net manager for the required location routes according to the previously specified protocol (`check`). If the routes are not available during the time windows of the plan, an alternative route is considered by backtracking to step 3.; otherwise the location routes are allocated (`book`). This process iterates until either a valid plan is found or the retry-bound is reached. In the later case, the new agent is deleted and an error message is raised.

(5) The new agent starts to execute its plan when the time is up.

Agent management The agent management of the TCS/MAS system is quite straightforward as it does not require a complex set-up or agent initialization procedure because the agents are modeled as threads within a single Oz process. Still, the communication system needs some mechanism to manage the peer identification process in an interaction. Therefore, we have implemented an agent directory service as shown in Figure 5.31. The communication unit of a newly created agent registers itself with the agent directory service and is assigned a unique identification number that is used in any message. Whenever an agent wants to send a message to another agent, the communication unit uses the agent directory service to obtain a handle for the addressee that is used to put the message in the in-queue of the other agent. For the sake of efficiency, the communication unit stores this handle in a peer list so that it can avoid to query the agent directory service twice for the same agent.

Communication model The communication model of the TCS/MAS system is built upon the method invocation process provided by the Oz system. However, since each agent has its own thread of execution, it is necessary

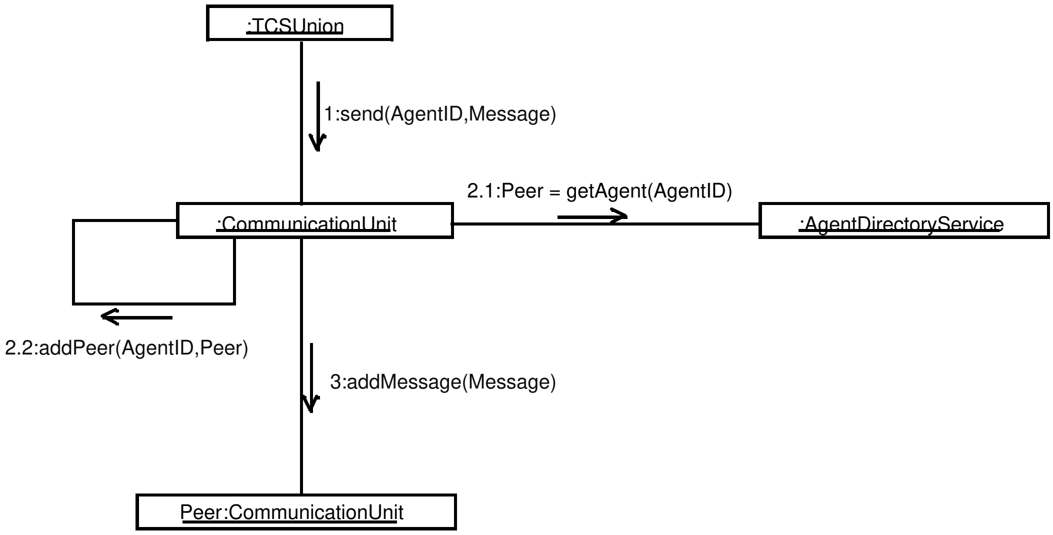


Figure 5.31: Message Passing in TCS/MAS

to de-couple the sending and the receiving thread. The resulting architecture of the communication sub-system is shown in Figure 5.32. The sending agent passes the message to its communication unit that is responsible for finding the receiving agent within the system. This is achieved by using the agent directory service provided by the agent management system as described above. Then, the communication unit asynchronously adds the message to the input buffer of the receiving agent and resumes the control flow of the local thread of its parent agent. The communication unit that has received the message, on the other hand, searches the currently active protocol execution units for the addressee and forwards the message to it. Then, the message is added to the input buffer of the matching protocol execution unit that handles the message within its own thread. If no matching protocol execution unit is found, an exception is raised.

Applying this scheme results in a flexible message passing mechanism that on the one hand de-couples the communicating thread and on the other hand allows an agent to be engaged in several ongoing interactions at the same time.

Agent architecture The architecture of the agents in the TCS/system will be developed in the next section.

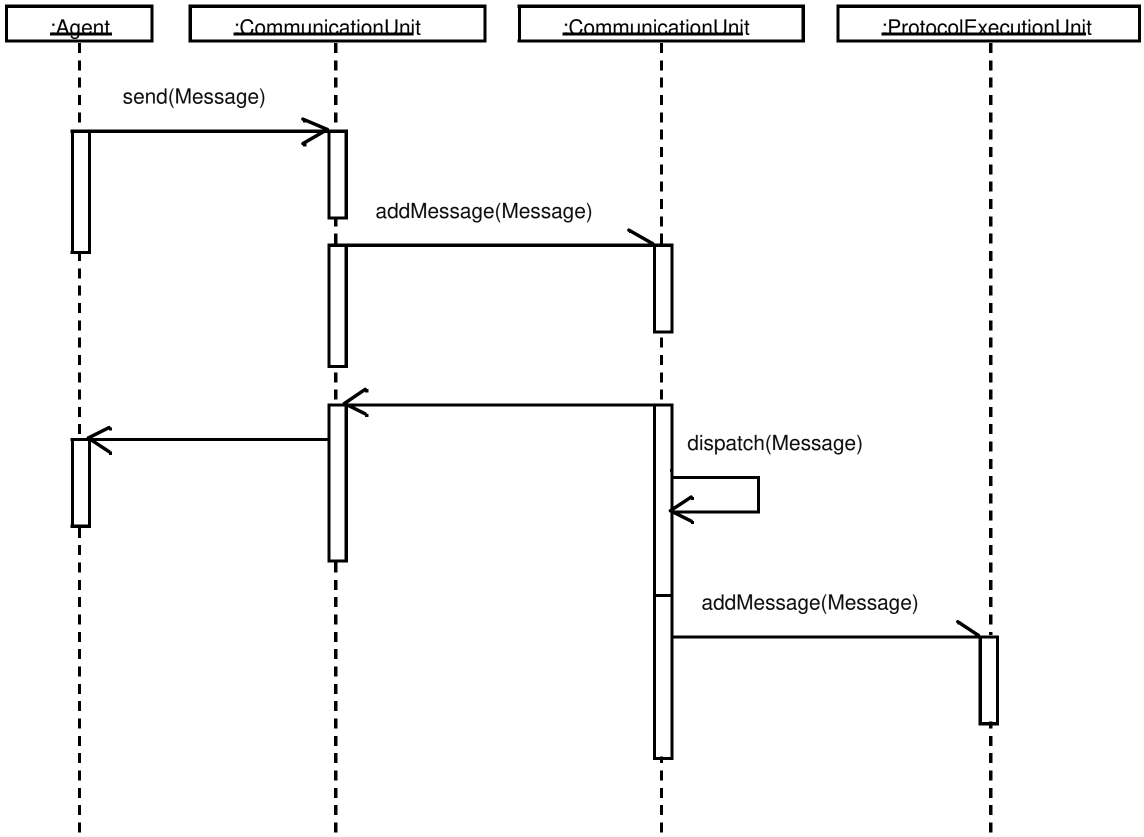


Figure 5.32: Message Passing in TCS/MAS

5.7.2.2 Agent Architecture

In order to summarize the requirements of the agent architecture of the TCS/MAS agent we will use the characterization scheme that was presented in Section 5.7.1.

Reasoning capabilities The TCS/MAS agents must have a planning unit to find their way through the railroad network and to detect route sharing possibilities with other agents. Furthermore, the agents must encompass some sort of reactivity when it comes to the plan execution simulation and failure handling.

Resources limitations An individual agent within the system has only limited computational resources because of the expected size of of the target system with approximately 5000 agents running in a single Oz process. Therefore, the agents must not be too complex as efficiency is a crucial factor due to the nonfunctional requirements given in Section 5.2.2.2.

Control flow The control flow within an agent is fixed over time and there is no need for a flexible, explicit representation of the control flow. However, an agent can potentially engage in several interactions at the same time and will also need to monitor the plan execution process while negotiating with other agents. Therefore, the agent needs a parallel action execution model in its architecture.

Knowledge Representation The major knowledge structures of the agents are the plans that hold the nodes, travel times and coupling activities of the agents. These plans are likely to change frequently and they must be interpretable by humans. Therefore, the plans are explicitly represented and kept in the knowledge base that must be provided by the agent architecture.

Autonomy The degree of autonomy of the agents varies in the different tasks they perform. In the optimization activities, the degree of autonomy is quite low as the agents only react to a sharing offer by a newly created union. However, things are different in the case of the plan execution facilities. Since failures of location routes must be handled by the agents, the re-planning process offers them the possibility to autonomously adapt their current plan whenever its is needed. Still, the agents do *not* implement pro-active behavior.

User interaction There is very little user interaction with an agent and thus the user interface is limited to presenting the plan and the current coupling state to the user and to allow the user to trigger plan execution.

Temporal context The lifetime of the agents ranges over a few hours and is therefore rather short. However, the system must provide some persistence mechanism to allow the user to store and retrieve schedules generated by the system.

In the TCS/MAS system, we have used the InteRRaP agent architecture (MÜLLER, 1996B) that provides most of the above features and that was also successfully used in the TELETRUCK (BÜRCKERT ET AL., 1998) system which shares some similarities with the TCS/MAS system.

InteRRaP The InteRRaP architecture is a generic agent architecture for situated agents that integrates reactive behavior and deliberation. The architecture

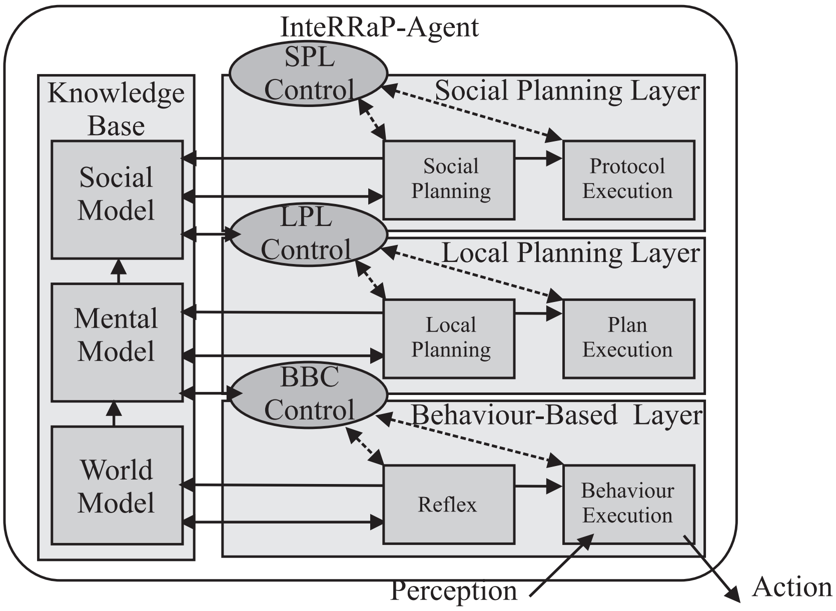


Figure 5.33: InteRRap

was designed for agents that exist within multiagent systems and thus some emphasis is on the communication aspect.

As depicted in Figure 5.33, `InteRRaP` is a layered architecture that consists of three layers, each consisting of concurrent processes:

Behavior Based Layer (BBL) This layer implements the reactive behavior of the agent, i.e. this layer reacts to external requirements without any explicit reasoning, thus it reacts very fast.

Local Planning Layer (LPL) This layer performs the planning process of an individual agent, it is also responsible to monitor the plan execution of the agents current plan.

Social Planning Layer (SPL) This layer is responsible for the coordination with the other agents within a multiagent system. The coordination with the other agents is achieved with explicit negotiation protocols.

All layers run concurrently, the intra-agent coordination between the three layers is achieved via the knowledge-base. The knowledge base is conceptually divided into three layers (world model, mental model, social model), but each computational layer has access to the knowledge on every level of the knowledge base. The conceptual discrimination, however, allows for a clearer design

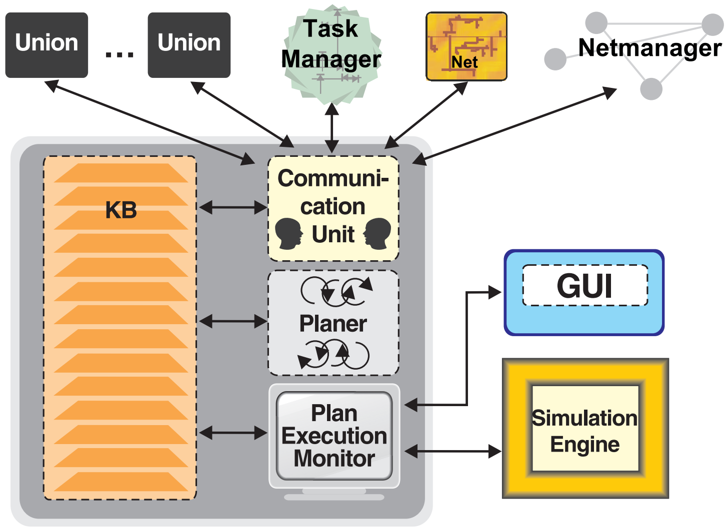


Figure 5.34: Union Agents

because most of the information stored in the knowledge base can be associated with a particular layer.

The InteRRaP architecture offers a generic framework for agent design that must be instantiated for the particular needs of a concrete scenario. Usually, some aspects of the generic framework are more interesting in a given scenario than others; the instantiation for the problem addressed in this dissertation is presented in the next paragraph.

Union Agent Architecture The union agents implement the previously introduced generic model in the context of the specific scenario. The emphasis of the instantiation is on the local planning layer and the social planning layer, the behavior based layer is not as important right now. However, in a later version of the system with a more realistic plan execution simulation, the BBL will become important as well.

A functional decomposition of the relevant parts of the union agents is shown in Figure 5.34 and the embedding of the TCS Unions in a broader context is shown in Figure 5.35.

Communication Unit (CU) The communication unit is responsible for executing the negotiation protocols between the agents. These protocols are used for the local planning process, to optimize the local plan of each agent and to handle the location route allocation process. The protocols

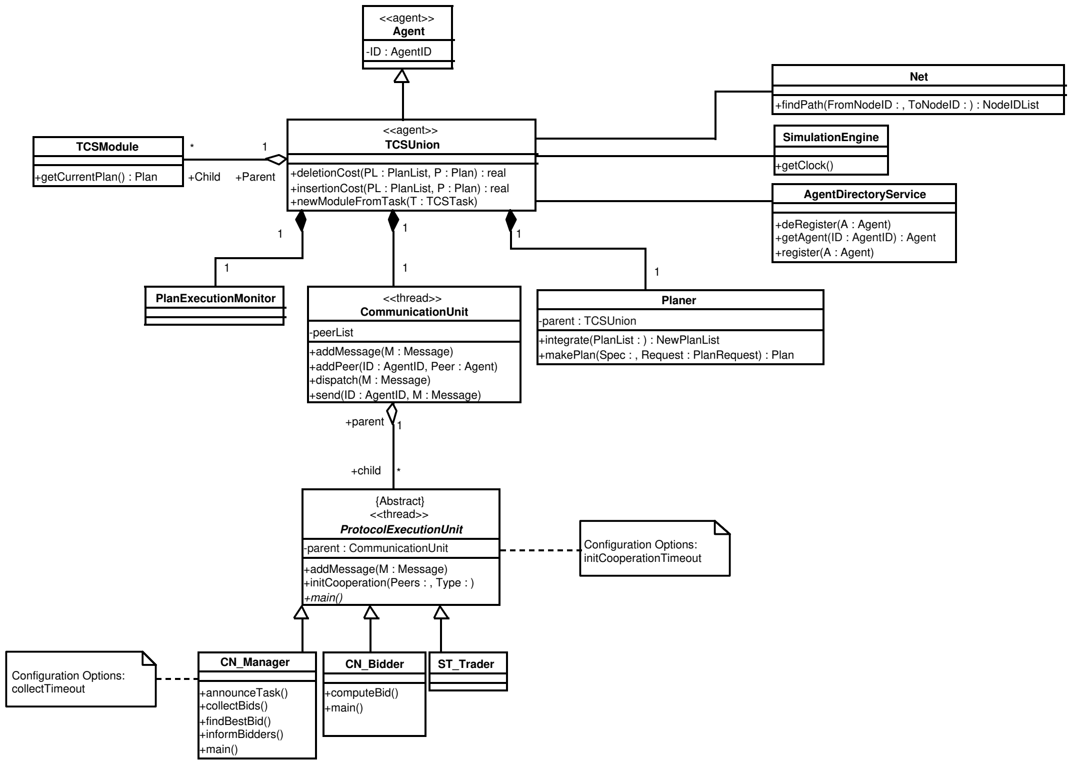


Figure 5.35: Union Agents (UML)

for the plan optimization steps will be explained in detail in the following section.

Planer The planer uses the shortest path service provided by the net object and the route allocation protocol of the net manger to find the local plan for an agent.

Plan Execution Monitor (PEM) The plan execution monitor monitors the execution of the current plan of an agent and adapts the plan in the case of external events, e.g. location route failures etc.

The arrows between the agent and the other parts of the system shown in Figure 5.34 indicate the connections of an agent to the outside world. Each arrow between the communication unit and another agent represents a logical connection that is used to execute the negotiation protocols between the agents. The connections to the simulation engine, the graphical user interface and the net object, on the the other hand, are achieved by method invocation within the object system.

In this section, we have outlined the system architecture of the TCS/MAS system and discussed the instantiation of a generic agent architecture to the specific scenario. In the next section, we will develop the final view of the view system for multiagent systems as outlined in Section 4.2.

5.8 System View

The main idea of the general view system is to structure the features of the target system according to their conceptual and logical links. However, some aspects of the target system are spread all over the system making it difficult to assign them to a particular view. Therefore, we introduce the System view that covers the aspects that affect multiple views or even the system as a whole.

5.8.1 General

5.8.1.1 User Interface Design

In this section, we will discuss some issues that are related to the user interface of the target system. Today, the main interface between a program and the user is usually a *graphical user interface (GUI)* and so we will limit the scope of this section to GUIs only. The user interface does not only play an important role with respect to the intended use of the system, but it is also important for the designer to know, that in most applications, more then 50% of the total system size are dedicated to the UI. However, due to limited space, we can only briefly mention some basic aspects and direct the reader to the literature for more information.

In the case of a multiagent application, the user interface can serve two major purposes. Firstly, it can manage the *task-specific user interaction* which is to request inputs from the user and in turn to present the results. Secondly, the user interface can be used to *monitor and manipulate the system activities* of the multiagent system. The designer (and later the user) must be able to follow the systems activities. The concurrent, distributed approach to problem solving makes this a difficult task that must be carefully executed. Monitoring the system activities is also of vital interest for the development phase of the system with respect to debugging. For monolithic systems, debugging features are supported by the programming language or a development environment; in the case of multiagent systems, however, only little or no support is given by existing languages and environments. Thus, either standard agent development frameworks such as ZEUS (NDUMU ET AL., 1999) must be used to visualize the system activities or the system designer must develop and integrate the facilities to support tracing and debugging into the design.

Despite the requirements of a particular class of applications, however, any user interface should comply with the following principles (GALITZ, 1997).

Clarity The entities of the GUI should be clearly related to the entities they represent in the real world and they should be given unambiguous names or icons.

Comprehensibility The user interface should be intuitive to use and provide an understandable access to the system functions. The key questions that describe this property are *What to look at?*, *What to do?*, *When to do it?*, *Why to do it?* and finally *How to do it?*.

Consistency An individual entity should have the same representation even if it occurs in different contexts and actions should always lead to the same results in order to guarantee some sort of predictability to user. Furthermore, the windows, dialogs etc. should have a consistent layout and appearance throughout the entire user interface.

Directness The user interface should provide a direct and intuitive access to accomplish the tasks. For example, complicated parameter settings in several menus before a function can be activated should be avoided.

Control It is important to design the interface in a way the provides the user with the feeling the he or she controls the behavior of the system and not vice versa.Thus, the system should query the user before taking action and it should keep the user informed about ongoing computations, loading process etc., for example by showing an hourglass whenever a longer action is launched.

Thus, a good user interface reflects the needs and capabilities of the user, obeys physical constraints of the hardware and conforms with existing standards. Our recommended iterative design process for the user interface consists of seven steps as shown in Process Model 7.

The seven steps of Process Model 7 are independent of a particular programming language or window system and can thus be used to guide the interface design on a very general level of abstraction. In order to speed up the development of the user interface, it is highly recommended to make use of existing software libraries such as Tcl/Tk (OUSTERHOUT, 1994) or Gecco (KNECHT, 1996) in order to benefit from off-the-shelf components for standard user interface elements.

Process Model 7 MASSIVE User Interface Design

(1) **Know your user** In this step the user groups are characterized according to their experience, the estimated use frequency, their skills (eg. typing or other input devices), the available amount of training, their motivation etc. This characterization will help the designer to develop a general idea of the user interface. It is, for example, a completely different task to develop a user interface for a novice that uses the system occasionally or for an expert who is familiar with similar applications and who regularly uses the system.

(2) **Relate to the system** The initial idea of the interface must then be related to the system that it is indented to represent towards the user. Therefore, the designer identifies potential points-of-interaction between the GUI and the underlying system. These points-of-interaction are all features that can be visualized or manipulated by the user.

(3) **Check standards** Before the real design process of the particular user interface starts, the designer should check its initial idea against existing standards and the constraints they may impose on the target interface. This step is important in order to avoid a user interface that is not generally accepted by the user community. However, if the entire target system is a customized application, deviations from the standard may be tolerable.

(4) **Define menus** The menus that usually appear on top of the application window define the overall structure of the GUI as they are usually the first entry point for the user. The menus should be related to functional groups within the system and have speaking names in order to enable the user to relate the menu titles and entries to particular functions of the system.

(5) **Select windows** In this step, the windows that represent different aspects of the system are being built. First of all, the designer must decide which information should be grouped together in a single window and then the following steps are executed for each window:

  (5.1) **Select presentation techniques** The most appropriate presentation technique, eg. textual, graphical or audio representation, is chosen for each of these groups and their elements.

  (5.2) **Select the appropriate screen-based controls** The control elements include the entire palette of tools offered by most existing window system interfaces, eg. text inputs, slide bars, list pickers, etc.

  (5.3) **Create the layout** In this step, the presentation and control elements are arranged within the window.

(6) **Create a help system** The help system is an important issue in a user interface when the interface reaches a particular level of complexity. The designer should provide two types of help to the user, the first one is a *general help system* that can be queried for specific topics by the user and the second is *context-specific help* that is activated by the user on-the-fly eg. by pressing a mouse button while the mouse pointer is over an input field.

(7) **Evaluate the interface design** The user interface is ideally checked by selected users if they are available. If the current version of the user interface fails in one of the aspects described above, the entire process is re-iterated from step 1.

Although user interface design is probably the most important and most extensive view that covers system-wide properties there are still other aspects probably not as exposed as the user interface that affect the system as a whole. These facets will be discussed in the subsequent sections.

5.8.1.2 Exception Handling

This aspect of the System view describes the exception handling policy of the target system – a feature that is usually spread all over the entire system and that effects almost every part of the target system. However, the term exception is usually used in a lot of different contexts as the following definition shows.

DEFINITION 5.8.1 (EXCEPTION (LANG AND STUART, 1998)) *An* **exception** *is the union of* error, exceptional case, rare situation *and* unusual event.

In order to provide a structure for the different aspects that are covered by the term, the following categorization was suggested by (COX AND GEHANI, 1989).

- *Software* or *design errors* are caused by implementational mistakes in the software, eg. dividing by zero, array index range errors, incorrect loop conditions etc.
- *Hardware errors* are the result of failures of the underlying hardware such as memory leaks, sensor failures etc.
- *State errors* occur if the systems model of the environment is inconsistent with the actual state of the environment; this kind of error is often found in robotic applications.
- *Timing errors* can occur only in real-time systems and are caused by the violation of timing constraints or resource (processor, memory) overload.

In the System view, however, we are only interested in exceptions of the first class and how to handle these design errors. In order to describe the error handling strategy of the target system, it is often useful to characterize the indented mechanism according to the characterization scheme given below.

Scope The scope of the error handling mechanism can either be *local* or *global*. A local strategy aims at the individual components of the system and specifies the error handling activities from the individual point-of-view.

A global strategy, on the other hand, introduces a central authority to which all errors are reported and that handles the exceptional situation according to a given plan. Multiagent system seem to naturally suggest a localized exception handling scheme, however this can make it sometimes difficult to cope with temporal information eg. when the designer needs to detect the exact temporal ordering of exceptions during the debugging phase. In this case, a global scheme can significantly reduce the development cost although the idea of a central authority opposes the basic multiagent idea.

Purpose The purpose of the exception handling process can be error *detection* or error *recovery*. The first case is easier to handle as it simply requires some sort of notification mechanism to indicate the presence of an exceptional situation to the designer or the user. Error recovery is usually much harder to achieve as it requires a thorough analysis of the current system state and explicit knowledge of how to handle a particular failure. However, in certain types of multiagent system, this type of exception handling will have to be considers as a simple system shut-down may not be an acceptable behavior.

Technology The technology aspect of the exception handling mechanism, finally, deals with the concrete implementation of this mechanism. A *language-based* approach uses the constructs of the underlying programming language to implement the exception handling strategy whereas an *operating system-based* approach makes use of services provided by the platform on which the target system is running. The choice of the best technology is a difficult matter that requires to weight several factors according to the requirements of the target system. While a language-based approach is usually easier to use and is mostly platform independent, it nonetheless depends on the expressive power of the target language which might be too limited for most multiagent applications. An operating system-based approach, on the other hand, is usually more platform dependent but also usually provides more flexibility to the designer.

Finding the exception handling strategy that is suited best for a particular target system is not easy and may have consequences for the entire system. Thus, it is usually a good idea to consider this aspect quite early in the development process, ideally before the actual implementation is started.

5.8.1.3 Performance Engineering

In this section, we will discuss some ideas that deal with performance aspects of the target system and we will present a micro process model for the Perfor-

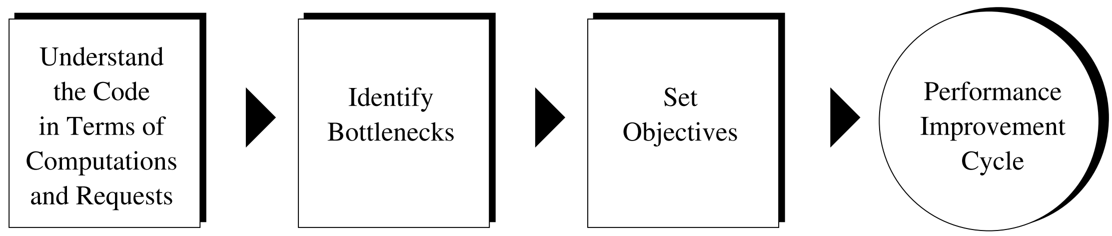


Figure 5.36: The Performance Engineering Process

mance Engineering process as defined below.

DEFINITION 5.8.2 (PERFORMANCE ENGINEERING (RATIONAL SOFTWARE, 1999B))
Performance Engineering *is a method to identify and reduce or eliminate performance problems during the software development cycle after the code has been designed and developed.*

The Performance Engineering process shown in Figure 5.36 is summarized in Process Model 8. The fundamental step in this process is to identify the bottlenecks that are responsible for performance losses in the target system. Some of the major causes for bottlenecks are listed below.

Useless computation Useless computation is often the result of program changes that make parts of the original code obsolete without removing the then unnecessary code fragments. Another source of useless computation are default computations that are executed even if they are not required, for example opening connections to remote agents by default can have severe impact on the systems start-up time.

Re-computation This bottleneck is the result of computing results although the could be cached for later use. The following example shows a very simple case of re-computation

```
if X.getRow() != MAXROW then
    StartRow == X.getRow() + 1;
    ...
end
```

In the example, the `getRow` operation on object `X` is called twice although it could have easily been cached as follows

Process Model 8 Performance Engineering

(1) **Understand** In the first step of the Performance Engineering Process the designer must develop a "feeling" for the runtime behavior of the application. This is best done by dividing the application into several phases (eg. initialization phase, input passing, etc.). Then the designer can decide, according to the separation of the runtime behavior, which phases are the most time consuming and focus the attention on these phases as improvements in these phases are likely to yield the highest gain in performance.

(2) **Identify** The goal of this step in the Performance Engineering Process is the identification of potential *bottlenecks* in the selected phases of the application. The sources of bottlenecks are manifold but is still possible to identify some prototypical classes that cover most existing bottlenecks.

(3) **Set Goals** When the potential bottlenecks are identified, the designer must set quantifiable goals in order to prioritize the bottlenecks according to their relevance. Several criteria for this are possible, the most obvious being the potential performance improvement. However, the cost (effort) to remove a particular bottleneck should always be weighted against any potential gain.

(4) **Performance Improvement Cycle** The goal of this, most important step in the entire process, is to isolate and eliminate a particular bottleneck. The Improvement Cycle as shown in Figure 5.37 consists mainly of six activities.

First of all, the test for a particular bottleneck must be carefully *planned* in order to enable the designer to focus on the performance aspect under consideration and to exclude any effects that might have influence on the result. Next, the test is *executed* and the performance of the program is *measured*. If the *evaluation* of the test results show, that the performance is already satisfactory according to the previously defined quality standards, the Performance Improvement Cycle can be aborted. If the performance is below the defined measure, the code is *changed* in order to remove the bottleneck and the test is *executed* again. If the next *measurement* shows that the code change has yielded the desired effect, the cycle is aborted. If this is not the case, the cycle starts again and repeats until either the performance goal is met or the effort exceeds a defined amount.

```
Tmp = X.getRow();
if Tmp != MAXROW then
    StartRow == Tmp + 1;
    ...
end
```

However, this is a rather trivial example that is often detected and removed by the compiler, but it can nonetheless have some impact on the overall runtime behavior eg. if the `getRow` function is computationally expensive or when the entire code sequence occurs within a loop.

Generally speaking, the higher the effort to compute a particular result and the more often it is needed, the higher is the performance gain

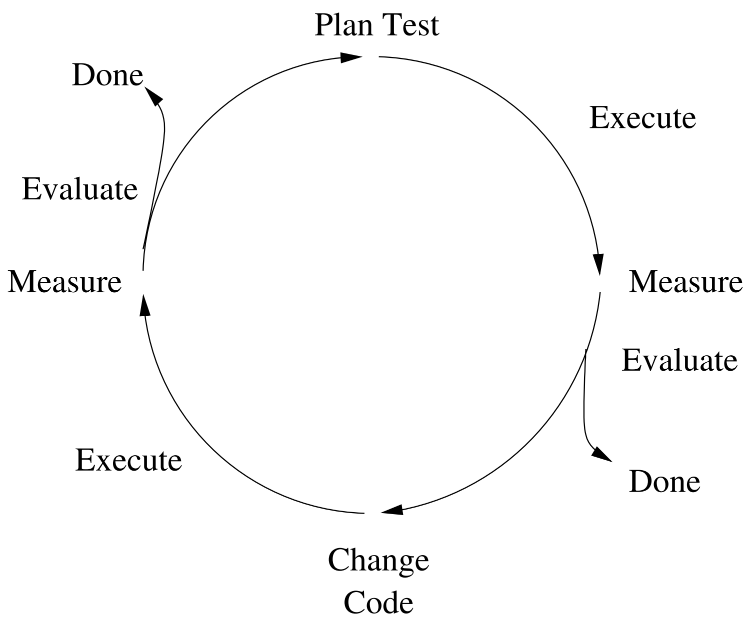


Figure 5.37: The Performance Improvement Cycle

through caching.

Waiting for service requests to complete Whenever a program requests a service from the operating system, it is typically blocked until the request is completed. Requests to the operating system are quite frequent in any computer program and consequently, some attention should be paid to these calls and how they can be reduced or transformed such that they are less vulnerable to external effects.

Prominent examples for this kind of bottlenecks are file access and memory management. In the first case, for example, the designer can make sure to be independent from network delays while accessing a file on a file system that is mounted via NFS if the file is read once and then kept in memory for further fast access. Obviously, this does only yield performance gains if the file is accessed more then once. In the case of memory management, it is sometimes useful not to rely on the memory management of the operating system. Allocating a large block of memory and then organizing the memory management locally is often a valuable alternative to the service provided by the operating system. However, even doing so does not prevent the application from being delayed because of page faults and heavy swapping. If this occurs, other mechanisms to speed up memory access must be found.

Wrong or missing assumptions about the runtime system This kind of bottleneck can often be found in the use of function parameters or local functions. Whenever a data structures is passed to a function using the call-by-value mechanism, the entire data structure must be copied on the stack and back again when the function is done. In the case of large structures, this copy operation can take a long time and additionally slow down memory access in the function body. Therefore, any call-by-value with a large data structure should be replaced with a call-by-reference even if the data structure is not changed in the function.

Local functions are a similar problem as the are often generated on the heap at runtime. This takes additional time and memory and should be avoided in frequently called function. We will discuss the impact of eliminating local functions in Oz programs in the case study in the next section.

Non-scalable algorithms or data structures This is a performance killer that is often found in applications that were developed and tested for small example data and that fail to work with the real operational data. A simple example is the following: assume, an agent has a list in which it stores all its acquaintances and that it looks up by performing a linear search whenever it sends a message and that is updated by appending a new entry whenever it receives a message from a formerly unknown agent. As long as the system is small, the effect of this list search and update mechanism can be neglected. However, if the system is scaled up to several hundred or even thousands of agents, this scheme will lead to performance losses that could be avoided by using a more efficient search strategy and/or a better data representation.

In this section, we have presented a general-purpose micro process model for Performance Engineering. However, Performance Engineering is not necessarily applied to the entire system, but it can also be applied to specific parts of it (SMITH, 1997). Furthermore, Performance Engineering is closely related to Refactoring (FOWLER, 1999) and some of the micro process models that are suggested for the refactoring process can be used for Performance Engineering as well.

5.8.1.4 Deployment

DEFINITION 5.8.3 (SOFTWARE DEPLOYMENT (HALL, 1999)) **Software deployment** *is the process that covers all of the activities performed after a software*

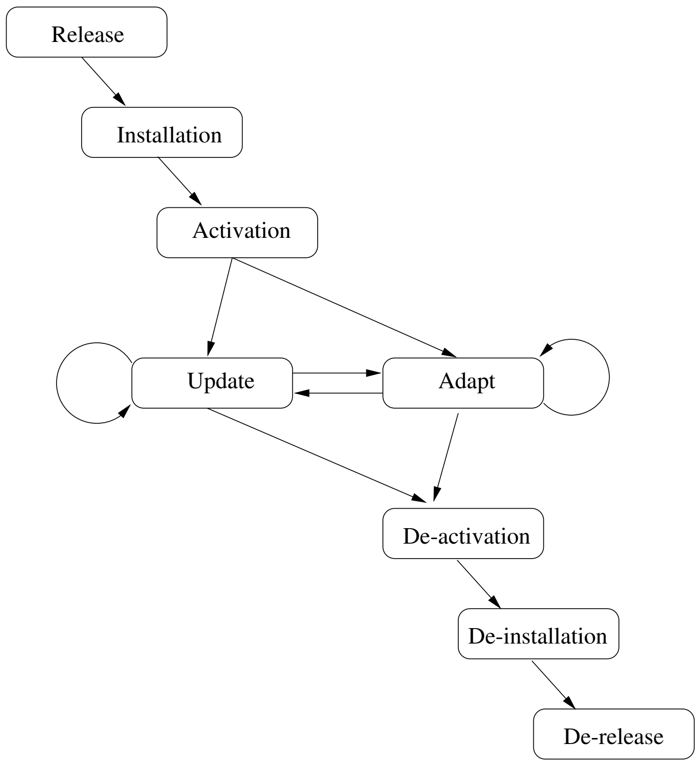


Figure 5.38: Deployment Life cycle

system has been developed.

The complexity of the deployment process depends on the complexity of the software and the required system environment. Planning of this process is essential for a successful implementation of the target system at the user site.

The deployment life cycle of a software system consists of eight steps as depicted in Figure 5.38.

Release This step is the interface between the development and the deployment process and includes all activities that are necessary to package the software system as well as the knowledge to set in operation at a user site.

Installation This step is usually the most complex activity because it must find and assemble all necessary resources. In this step the system as well as the external resources such as libraries, software packages etc. are either initially deployed at the user site or updated according to the required versions.

Activation The activation of a software system refers to the process of starting the participating components in order to get the system running. For simple software systems, this may require only to push a button or to enter a command line. More complex systems, however, may require more sophisticated, coordinated activities in order to bring all components into operation.

Update Updating am already installed software system means to modify it in order to add new functionality or to remove bugs. Updates are issued by the software provider and taken up by the clients.

Adapt The adaption of an installed system differs from an update in that the former is limited to local changes only while the latter refers to potentially all installed systems as a whole. Adaption thus refers to changes of the system at a particular site in order to adapt it to changes in the system environment.

De-activation De-activation is the inverse process to the activation of the system and refers to a controlled shut-down of all components involved in running the system.

De-installation When the system is no longer required at a user site, it must be removed from the site. This it not necessarily a trivial process because attention must be paid to not disturbing the system environment of the user site eg. by the deletion of shared resources. Thus, this step is not the process of undoing everything that was done upon the installation of system but it requires an analysis of the current state of the user site to detect dependencies of other software systems on resources that were installed with the target system.

De-release The final step in the deployment life cycle is reached when a system is regarded obsolete and it is no longer developed or supported by the manufacturer. This step is distinct from the previous one in that it does not mean that the system cannot be used any longer. Rather, the users are free to use the system but they should be aware that no further support will be available.

The deployment of a particular system is usually a highly individualized task and thus hard to capture in a general process model. The following generic deployment process model is thus described at a rather high level of abstraction and will need to be tailored according to characteristics of a specific project. Basically, the deployment process consists of five steps as shown in Process Model 9.

Process Model 9 System Deployment

(1) **Identify and characterize components**
The components that are relevant for the planning of the system deployment are for example the executable files, data sources, hardware devices, external (software) components etc. Each of these components is characterized for example by the site characteristics were it should be installed, the deadlines it must hold, its availability or a version specification. The goal of this step in the planning process is to obtain a complete picture of every entity that is related to the target system in one way or another.

(2) **Describe dependencies**
In this step, the dependencies between the components are explicitly modeled. A dependency is for example a "uses" relation between components or temporal precedences in the installation process that must be maintained. Note that dependencies can vary between different user sites.

(3) **Define Activities**
For each component, the activities that are necessary to set a component into the state that is required by a user site are specified. Examples for activities are the steps that must be performed to install a software library or the instructions to set up a particular hardware device.

(4) **Execute**
In this step, the previously defined activities are executed in the order imposed by the dependencies that were modeled in step 2.

(5) **Start**
In the final step of the deployment process, the system is set into operation at the user site.

This five-step generic deployment process covers the major parts of the deployment life cycle shown in Figure 5.38: the release of a particular system is described in step 1, the installation in step 4, the activation in step 5 and the update activity of the life cycle is captured in steps 1 and 4. Other activities in the life cycle are not directly covered but the can make use of the gathered information, eg. the de-installation will certainly use the component characterizations and the dependency information when removing a system from a particular site.

5.8.1.5 Summary

In this section, we have presented the view that models the features of the target system that are spread over different views or even the entire system. We have discussed aspects of the user interface design, exception handling, Performance Engineering and system deployment. All these aspects affect many different parts of the system and are thus collected in the System view.

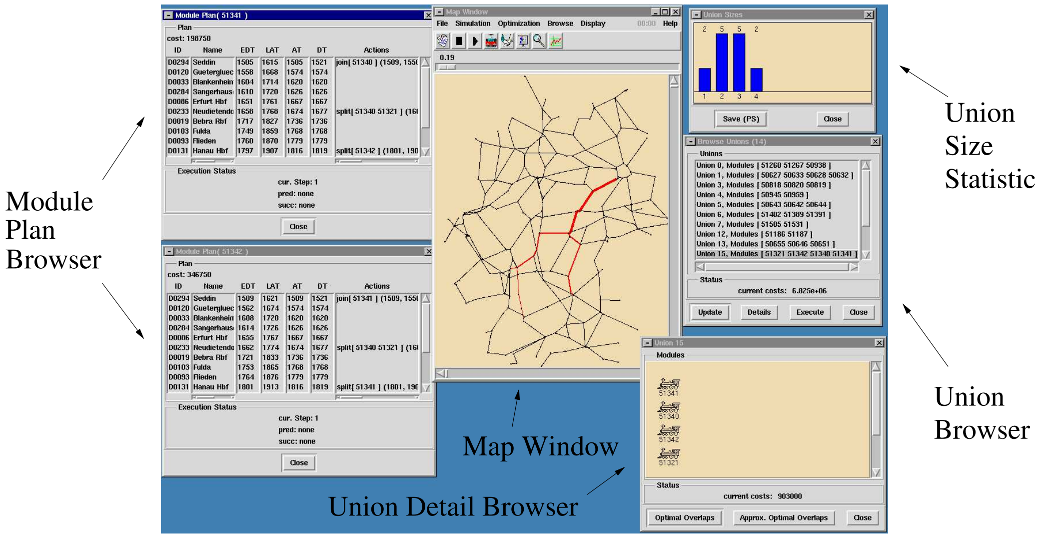


Figure 5.39: Elements of the TCS/MAS GUI

5.8.2 Case Study

5.8.2.1 Graphical User Interface

The graphical user interface of the TCS/MAS system supports the detailed inspection and monitoring of the ongoing computation as well as the off-line analysis of simulation data obtained in batch-mode of the system. The main elements of the GUI are shown in Figure 5.39 which contains a screenshot of the TCS/MAS system with number of monitoring and statistic elements.

Map Window The Map window shows the underlying railroad network and can be used to visualize the plan execution process of the unions. It serves also as the main entry of user interaction as it contains the menu bar that controls all major functions of the system.

Task Viewer The Task viewer is used to browse the task file that is currently loaded. For each task, the module identification, the module type, the module length, the source node, the destination node, the earliest possible departure time, the latest allowed arrival time and the time when the task is announced to the system are shown.

Union Browser The Unions browser shows the current union structure of the system, ie. which modules are connected in a particular union. The user can select one or more of these unions to check for the current state of the unions, eg. for the current coupling state of the union or the user can trigger the plan execution process for these unions to check the validity of their current plan.

Plan Browser The Plan browser is started from the Union browser window and allows for an even closer look at the plans of the modules that make up the union. For each module, the Plan browser shows a sequence of nodes where each of the nodes consists of the node identifier, the earliest possible arrival time in the node, the latest allowed departure time from the node, the scheduled arrival and departure times and the actions that must be executed in the node. For each action, the scheduled time window and the peer modules are also shown.

Statistics Viewer The Statistics viewer is a collection of tools that enable the user to examine various properties of the current schedule. The *Union statistic* shows the distribution of the unions in the system over the possible union sizes, the *Coupling Node statistic* uses the Map window to visualize the nodes in the system that are scheduled for coupling activities, the *Source and Sink Node statistic* also uses the Map window to present the nodes where the freight is picked up from or delivered to, and the *Location Route Monitor*, finally, exhibits the usage statistic for every location route that is used in any module plan.

Trading Monitor The Trading monitor can be activated by the user to scrutinize the hypothetical trading operations between unions as well as the trading match that is computed by the stock manager. Whenever the Trading monitor is active, the user can interactively decide whether a particular match is accepted or not.

Communication Monitor The Communication Monitor, finally, is used to visualize and inspect the messages that are exchanged between the agents. The user can see the messages that are sent by one agent to another and can click on a particular message to inspect the contents that are communicated.

5.8.2.2 Exception Handling

Because of the prototypical nature of the TCS/MAS system, we have refrained from implementing an elaborate exception handling scheme in order to simplify the design of the system at the expense of stability in case of unforeseen events. Thus, the exception handling mechanism that is implemented features only the basic requirements in that it is *local*, is used only for *error detection* and is *language-based*.

Locality means, that each entity performs its own error monitoring process and does not report failures of any kind to a central authority. This is not necessary as the goal of the exception handling process is only error detection and not the recovery from unexpected system states. The TCS/MAS system uses the exception notification mechanisms provided by the runtime system of Oz to detect design errors and to report them to the developer or user. It is then up to him or her to take the appropriate action to prevent the situation that lead to an error in subsequent activations of the system.

5.8.2.3 Performance Engineering

The initial performance of the TCS/MAS showed a promising behavior although the number of agents that were active in the system was quite high with approximately 5000 agents for the largest examples. Nonetheless, however, we were able to increase the system performance by a factor of six by performing two rather simple nonfunctional optimizations.

Using the profiler that comes with the Oz distribution, we were able to identify several bottlenecks in the actual implementation of the system. The first bottleneck was due to our lack of knowledge of the runtime system of Oz and the second bottleneck was a suboptimal design in a basic data structure.

Let us begin with the runtime behavior of Oz. Whenever a member function of a class contains a local function, the *closure* of this local function is constructed on the heap a runtime, ie. whenever the member function is called. The following example shows an example for a local function that is used to implement a loop - a method that is quite common in Oz programs.

```
class A
    meth findElem( List Key ?Result )
        FindElem = fun{ $ List Key }
                       ...
                     end
    in
        Result = {FindElem List Key}
    end
end
```

Thus, whenever `findElem` is executed, the runtime system builds the closure of `FindElem` on the stack and removes it after `findElem` is done. This mechanism does not cause any problems as long as `findElem` is not called frequently in the application.

In the TCS/MAS system, on the other hand, we had a small number of functions that used local functions and that were themselves used millions of times in the course of a system run. These functions were identified using the profiler and their code was re-written as illustrated below.

```
local
    FindElem = fun{ $ List Key }
                    ...
                  end

    class A
        meth findElem( List Key ?Result )
            Result = {FindElem List Key}
        end
    end
```

This rather trivial re-arrangement of code fragments brought a performance enhancement by a factor of four and thus by far justified the effort and the break of the principle of locality of computation.

The second bottleneck in the implementation was a combination of the use of a sub-optimal algorithm and re-computation and was caused by the class that encapsulates the network data of the underlying railroad network. The `Net` class maintains the list of links within the network and provides a member function that takes a link identification number as input and returns the matching link object to the function caller. In the first version of the TCS/MAS system, we used a small network of approximately 25 nodes to develop the basic planning algorithms in a controlled environment of limited size. Therefore, the list of links was relatively small and the member function that mapped link identifications to links was implemented as a sequential search over the link list. As the system was upgraded to the full network, however, the search algorithm was not reviewed with respect to the higher performance demand in a 350 node network and remained in place. Again, the profiler of Oz directed us to this function and the search process was replaced by a direct mapping that was held in a dictionary and the performance gain was about a factor of 1.5.

The rather simple nonfunctional optimizations of the TCS/MAS system clearly show, that it pays for the system designer to carefully monitor the systems runtime behavior and that large performance gains are not necessarily hard to achieve.

5.8.2.4 Deployment

Since the TCS/MAS system was designed as a prototype that was only to be run at the development site to obtain simulation data there was no need for extensive deployment planning.

The only relevant aspects of the system deployment is to ensure the installation of the correct versions of Oz (PROGRAMMING SYSTEMS LAB, 1999) and LEDA (MEHLHORN AND NÄHER, 1999). For development versions of the TCS/MAS system, Emacs (STALLMAN AND FREE SOFTWARE FOUNDATION, 1999) must be installed as well.

5.9 Summary

In this chapter, we have presented views as the core concept of the MASSIVE development method. Views allow the system designer to break down the target system into several projections that concentrate on particular aspects of the system and abstract away from others. The seven views of the MASSIVE method are summarized in Figure 5.40. The Task view focuses on what the system should do in terms of functional and nonfunctional requirements and basic problem solving capabilities without thinking in agent-oriented terms. The Environment view characterizes the environment of the target system from the developers and the systems perspectives. The Role view concentrates on a functional and physical grouping of the problem domain and is used to assemble a set of coherent roles for the agents. It also deals with the assignment of these roles to particular agents. The Interaction View describes the ongoing interactions between agents in terms of interaction protocols and provides a set of generic interaction schemes that can be used by the system developer. The Society view defines the social dimensions of the target system and describes the structure of the agent society. The Architectural view turns the features that have been modeled in the other views into the software design of the system and the agent architecture. The System view, finally, captures all aspects that cannot be assigned to a single view such as the user interface, the global error handling scheme, performance engineering aspects or system deployment.

The suggested product model represents our current experience with the development of multiagent system and is meant as an initial model that should be refined and extended in the sense of the institutional framework of an Experience Factory.

In the next chapter, we will demonstrate how the MASSIVE method was successfully applied in other problem domains with a focus on the product model

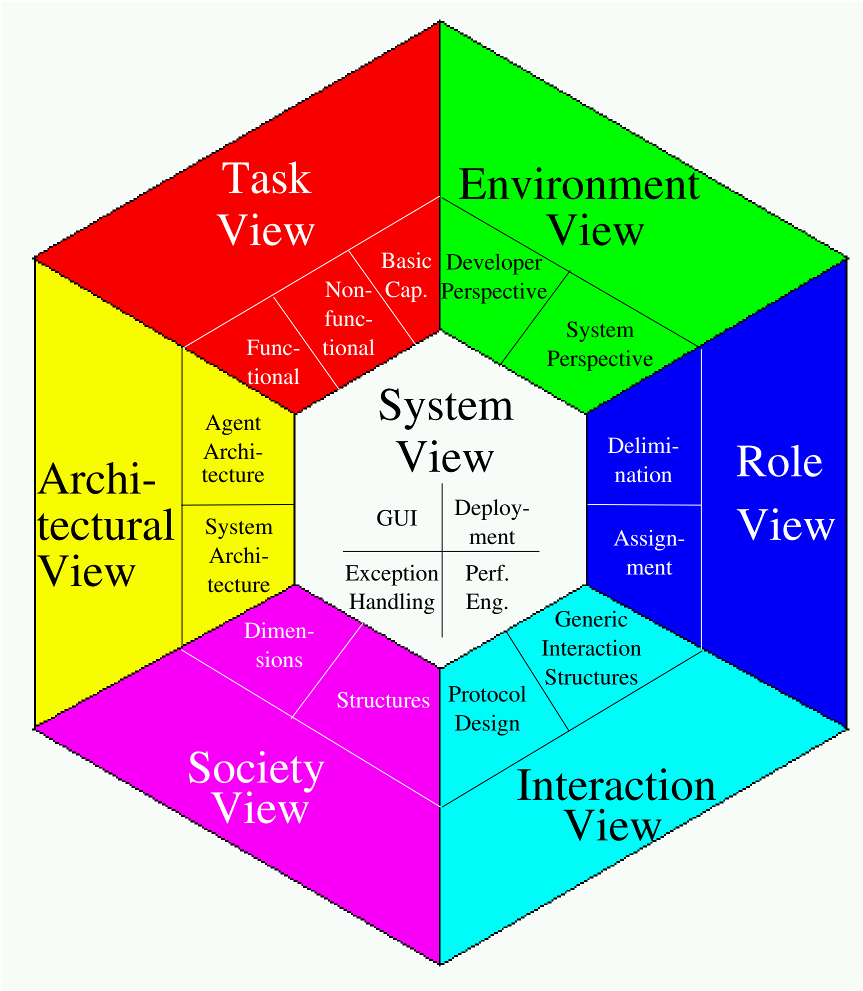


Figure 5.40: The MASSIVE View System

rather then on the process model.

CHAPTER 6

Further Case Studies

A DEVELOPMENT METHOD WILL NEVER BE ACCEPTED as an industrial standard if it cannot prove its validity in practice. The MASSIVE method is not a method that was developed in the laboratory and then transfered to actual projects, but rather it is derived from projects that were successfully carried out at the DFKI and elsewhere and that were analyzed after completion in order to find similarities in the product and process models. The advantage of this approach is that it provides further case studies that show how the method works and that demonstrate that the method can be used for a broad range of multiagent applications.

6.1 Personal Travel Assistant: Intermodal Route Planning

The Personal Travel Assistant (PTA) (FIPA, 1997) is a scenario that is defined by the FIPA Standardization Organization to evaluate various aspects of the standards. The agents in the PTA domain operate on behalf of their users and provide assistance in the pre-trip planning phase as well as during the on-trip execution phase. In order to accomplish this assistance, the PTA interacts with the user and with other agents that represent the available travel services. Besides the core competences such as configuration and delivery of trip planning and guidance services, the PTA also provides added-value services according to per-

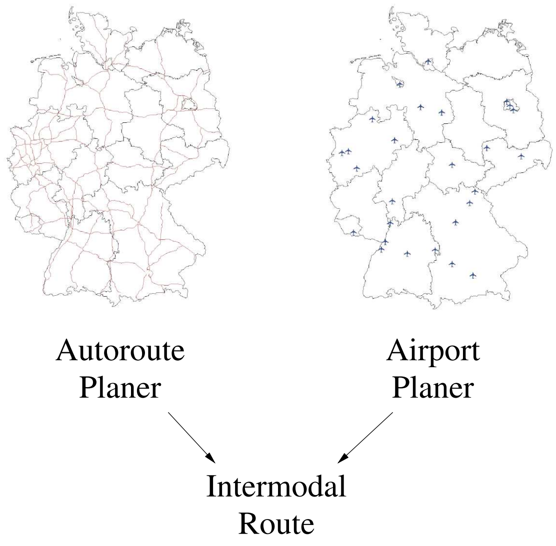


Figure 6.1: IMRP Example

sonal profiles, e.g. interests in sports, theater, or other attractions and events.

The focus of the services provided by the PTA, however, is on the basic requirements of the trip which is first of all to find the best combination of travel services that take the PTA user from his or her desired starting point to the destination. This route planning service is provided by a special agent, the *Intermodal Route Planner (IMRP)*. The IMRP is responsible for selecting a group of transport carriers that provide the necessary services at the quality required by the user. Figure 6.1 illustrates the basic idea in combining the services of several transportation carriers such as car and plane into a single route plan that guarantees a smooth trip.

In the following sections, we will describe our solution for the intermodal route planning process that was developed in the MoTiV-PTA project (BAYRISCHE LANDESREGIERUNG, 1996). Due to the limited space, however, we will only be able to provide the basic ideas of the design that was used to implement the final system.

6.1.1 Task View

The Task view defines the basic concepts that were used to model the problem domain. We will give a declarative description of the task to be solved.

Functional Requirements First of all, we define the basic data structures that occur in the problem domain and that must be represented within the target system. The most important data structure for the formalization of the IMRP domain is used to represent the individual travel services. Each of these services operates on a graph structure in which the nodes represent the service access points and the weights of the links represent some abstract cost measure that is associates with traveling a particular link.

DEFINITION 6.1.1 (GRAPH) *A* **Graph** $G = (V, L, \alpha, \omega, \gamma)$ *consists of a set* V *of vertices, a set* L *of links, a mapping* $\alpha : L \to V, \alpha(l) = v$ *where* $v \in V$ *and* v *is a starting node of* l*, a mapping* $\omega : L \to V, \omega(l) = v$ *where* $v \in V$ *and* v *is a terminal node of* l *and a weight function* $\gamma : L \to \mathbb{R}$ *that assigns a weight to each link.*

l *is a link between two vertices* v_1 *and* v_2 *iff* $\alpha(l) = v_1 \wedge \omega(l) = v_2$*. The weight of the link is* $\gamma(l)$*. In the context of multiple graphs, let* V_G *and* L_G *denote the set of vertices and links for a graph* G*, respectively. In the following, we will use the terms vertice and node synonymously.*

Another data structure that is often used in the formalization is a list over a particular set of ground elements. We give the definition below to introduce the basic operations that can be performed on such lists.

DEFINITION 6.1.2 (LIST, LIST OPERATIONS) *Let* $L = [e_1, \ldots, e_n]$ *be a* **List** *of elements over a set* E*, ie.* $e_i \in E \quad \forall i \in \{1, \ldots, n\}$*. We write* $L = [H|T]$ *with*

- $H = e_1$
- $T = e_2, \ldots, e_n$

Let $LISTS(E)$ *be the set of all sets over* E*. The the following operations are defined on* $LISTS(E)$*:*

- $length : LISTS(E) \to \mathbb{N}$ *where* $length([e_1, \ldots, e_n]) = n$ *returns the* **length** *of a list.*
- $head : LISTS(E) \to E$ *where* $head([H|T]) = H$ *returns the* **head** *element of a list.*
- $tail : LISTS(E) \to LISTS(E)$ *where* $tail([H|T]) = [T]$ *returns the* **tail** *element of a list.*

Furthermore, we define

- $heads : LISTS(LISTS(E)) \to LISTS(E)$ *where* $heads([[H_1|T_1], \ldots, [H_n|T_n]]) = [H_1, \ldots, H_n]$

- $tails : LISTS(LISTS(E)) \rightarrow LISTS(LISTS(E))$ *where* $tails([[H_1|T_1], \ldots, [H_n|T_n]]) = [[T_1], \ldots, [T_n]]$

After these basic data structure definitions, we can now turn to the actual formalization of the concepts of the IMRP domain. First of all, we formalize the services that are restricted to individual travel services.

DEFINITION 6.1.3 (UNIMODAL MAP) *A* **unimodal map** *is a graph* G *according to definition 6.1.1,* $UMAPS$ *is the set of all unimodal maps.*

We define a predicate $valid : UMAPS \rightarrow \{true, false\}$ *on the set of unimodal maps that decides the* **validity** *of a map in a given context.*

The validity of a route is a personalized function that can be used to express user preferences with respect to a particular travel service. We will not define this function as it is only an interface to higher level functionality of the PTA that is not of interest in the IMRP domain. Next, we define how a particular route is represented such that it can be uniquely identified within a intermodal route.

DEFINITION 6.1.4 (UNIMODAL ROUTE) *A sequence* $R = (v_1, \ldots, v_n)$ *is a* **unimodal route** *in a map* M *iff*

- $v_i \in V_M \quad \forall i \in \{1, \ldots, n\}$ *and*
- $v_i \neq v_j \quad \forall i, j \in \{1, \ldots, n\} : i \neq j$ *and*
- $\exists l \in L_M : \alpha(l) = v_i \wedge \in \omega(l) = v_{i+1} \quad \forall i \in \{1, \ldots, n-1\}$

We call a tuple $((v_s, \ldots, v_d), G)$ *with* $G = \sum_{l \in L(v_s, \ldots, v_d)} \gamma(l)$ *a* **weighted (unimodal) route** *and* $U = (M, R, G)$ *a* **named weighted (unimodal) route** *in* M*.*

Furthermore let $Routes(M)$ *denote the Set of all Routes in* M *and the set of all links of a particular route is* $Links : Routes(M) \rightarrow L_M^n$ *with*

$$Links(R) = \{l | l \in L_M \wedge \exists v \in V_M : (\alpha(l) = v \vee \omega(l) = v)\}$$

$\Gamma : Routes(M) \rightarrow \mathrm{IR}$ *is defined as*

$$\Gamma(R) = \begin{cases} MAXWEIGHT & if \quad v_s = v_d \quad or \quad R = () \\ G & otherwise \end{cases}$$

Note that we have defined the weight of a route to be $MAXWEIGHT$ in the case that the start and the goal node are the same. This trick allows us to simplify the algorithms given below as it automatically rules out empty routes in an intermodal route without an extra check simply because of the high costs that are associated with any intermodal route that contains empty unimodal routes.

Now that the individual travel services have been formalized, we must define how the PTA user can switch from one travel modality to another. Thus, we must define how we describe access points that are shared by several travel services.

DEFINITION 6.1.5 (JUNCTION POINTS) *The* **junction points** *between two maps* M_1 *and* M_2 *are defined as*

$$J(M_1, M_2) = \{(v_1, v_2) | v_1 \in V_{M_1} \wedge v_2 \in V_{M_2} \wedge samePlace_{M_1M_2}(v_1, v_2) = true\}$$

Where the predicate $samePlace_{M_1M_2} : V_{M_1} \times V_{M_2} \rightarrow \{true, false\}$ *decides whether two nodes occupy the same position or not.*

Furthermore let

$$J_{M_1M_2}(v) = \{w | v \in V_{M_1} \wedge w \in V_{M_2} \wedge (v, w) \in J(M_1, M_2)\}$$

be a function that returns all junction points for a given node and fixed source and destination maps and let $Alias : V \rightarrow V$

$$Alias(v) = \{w | w \neq v \wedge \exists M_1, M_2 : samePlace_{M_1M_2}(v, w) = true\}$$

be a functions that returns all nodes that are equivalent (with respect to the $samePlace$ *predicate) to a give node. Note that* v *itself is contained in this set as well.*

The actual decision whether two nodes of different maps represent a junction point is encapsulated in a specific predicate $samePlace$ in order to make it easier to explicitly define junction points. A very straightforward definition of the predicate is to base it on the Euclidean distance of two nodes, but more sophisticated schemes can be implemented as well.

In the next steps in the formalization of the IMRP domain, we will define how several individual travel services can be combined in a single intermodal route. Therefore, we will first of all combine all individual travel services in a joint map in order to ease the reasoning process over several services.

DEFINITION 6.1.6 (INTERMODAL MAP) *A* **intermodal map** *is a set of graphs* $(G_1, \ldots, G_n)$ *with*

- $L_{G_i} \cap L_{G_j} = \emptyset \quad \forall i, j \in \{1, \ldots, n\} : i \neq j$, *and*
- $V_{G_i} \cap V_{G_j} = \emptyset \quad \forall i, j \in \{1, \ldots, n\} : i \neq j$

Furthermore let $MAPS$ denote the set of all intermodal maps and let $Map : \bigcup_{G\in\{G_1,\ldots G_n\}} V_{G_i} \to UMAPS$ be a function that maps a node to the map that contains that node.

Now, we can express an intermodal route in terms of this underlying data structure.

DEFINITION 6.1.7 (INTERMODAL ROUTE) *Let $U = (M, R, G)$ be a named weighted unimodal route in M and define further*

- $first(U) = first(M, R, G) = first(M, (v_1, \ldots, v_n), G) = v_1$ *and*
- $last(U) = last(M, R, G) = last(M, (v_1, \ldots, v_n), G) = v_n$ *and*
- $V(U) = R$ *and*
- $M(U) = M$.

A sequence of unimodal routes $MMR = (U_1, \ldots, U_n)$ is called **intermodal route** *between v_1 and v_2 iff*

- $v_1 = first(U_1)$
- $(last(U_i), first(U_{i+1})) \in J(M(U_i), M(U_{i+1})) \forall i \in \{1, \ldots, n-1\}$
- $v_2 = last(U_n)$

Let $R(MAPS)$ be the set of all intermodal routes and $\Gamma : R(MAPS) \to \mathrm{IR}$ be defined as

$$\Gamma(MMR) = \begin{cases} MAXWEIGHT & if \quad MMR = () \\ \sum_i weight(U_i) & otherwise \end{cases}$$

The **normal form** $\downarrow: R(MAPS) \to R(MAPS)$ *of an intermodal route $MMR = ((M, R, G)_1, \ldots (M, R, G)_n)$ is defined as*

$$MMR \downarrow = ((M, R, G)_{i_0}, \ldots (M, R, G)_{i_m}))$$

with

- $i_0 < i_1 < \cdots < i_m$
- $G_{i_j} \neq \emptyset \forall j$

In the above definition we require for any intermodal route that junction points exist between any two successive unimodal routes that are combined in the intermodal route. We also define a normal form on intermodal routes in order to able to compare two intermodal routes and to rule out such routes that contain empty unimodal routes.

In the last step of the formalization, we will define how to express a request for an intermodal route in terms of the unimodal maps involved.

DEFINITION 6.1.8 (ROUTING SCENARIO) *A sequence* $(M_1, \ldots, M_n)$ *of maps is called a* **routing scenario** *in the set of unimodal maps* ($UMAPS$) *if the following conditions hold:*

- $v_s \in M_{i1} \wedge v_d \in M_{in_i}$ *for* $i = 1, \ldots m$ *and*
- $\exists v_3, v_4 : (v_3, v_4) \in J(M_{ij}, M_{ij+1}) \, \forall i, j : i \in \{1, \ldots, m\}, j \in \{1, \ldots n_i - 1\}$

Let $RS(UMAPS)$ *denote the set of all possible routing scenarios.*

The function $RS : V \times V \rightarrow RS(UMAPS)$ *returns for any two nodes* v_s *and* v_d *a list of possible routing scenarios* $((M_{11}, \ldots, M_{1n_1}), \ldots, (M_{m1}, \ldots, M_{mn_m}))$.

Furthermore we define $filter : RS(UMAPS) \rightarrow RS(UMAPS)$ *as follows*

$$filter((M_1, \ldots, M_n)) = \begin{cases} (M_1, \ldots, M_n) & valid(M_i) \quad \forall i \in \{1, \ldots, n\} \\ \emptyset & otherwise \end{cases}$$

$filter$ *is extended on a sequence of routing scenarios by defining* $filter : RS(UMAPS)^n \rightarrow RS(UMAPS)^m$ *as*

$$filter(RS_1, \ldots, RS_n) = (filter(RS_1), \ldots, filter(RS_n))$$

The RS function serves as interface to some higher level service that provides all possible routing scenarios between any two nodes of any unimodal travel services and the $filter$ function operates as interface to the user preferences just like the $valid$ predicate defined above.

Nonfunctional Requirements The first nonfunctional requirement for the target system is that its architecture and the services provided must be FIPA compliant. Furthermore, the answering time of the target system should lie within a range that allows the user the on-line planning of a trip, thus the answering time will typically range within a few minutes.

The final nonfunctional requirement is to evaluate two different forms of interaction, the first one of which is a central planning approach that features one intermodal route planner that uses the services provided by the unimodal planners and the other one being a distributed scheme where each unimodal planner can become an intermodal planner upon a user request.

Basic Problem Solving Capabilities Since the aspect of intermodal route planning is only a small part of the entire PTA domain, the basic capabilities that are needed within this sub-part do not spawn over a broad range. Basically, the only atomic problem solving capabilities that are required are the ability to perform unimodal route planning by either planning from first principles or by using an existing service and providing the result to the other agents in the system, and the ability to combine several unimodal routes into a single intermodal route.

The process of combining the services of the unimodal route planners into a single plan works as follows: At the first level of the planning process all possible routes between all alias-nodes for the source node and alias-nodes of the destination node are computed. Alias nodes exist iff the source or the destination node are junction points with another map. In this case, all routes between all combinations of alias nodes must be computed.

Input v_s, v_d

Algorithm

$$imrp(v_s, v_d) = \bigcup_{v_1 \in Alias(v_s) \times v_2 \in Alias(v_d)} imrp\_na(v_1, v_2)$$

for all $v_1 \in Alias(v_s)$ **do**
  for all $v_2 \in Alias(v_d)$ **do**
    $result := result \cup imrp\_na(v_1, v_2)$
  end for
end for
return $result$

Output A list of all routes between all alias nodes for v_s, v_d

At the second level, all routes for all routing scenarios between two alias nodes are computed according to the following algorithm:

Input v_s, v_d

Algorithm

$$imrp\_na(v_s, v_d) = \bigcup_{rs \in RS(v_s, v_d)} imrp\_sc(v_s, v_d, rs)$$

for all $rs \in RS(v_s, v_d)$ **do**
  route := $imrp\_sc(v_s, v_d, rs)$
  if $route \notin result$ **then**
    $result := result \cup \{route\}$
  end if
end for
return $result$

Output A list of all routes for all routing scenarios between v_s, v_d

In this view, we have formalized the IMRP domain in terms of functional abstractions that can be used by the subsequent views to develop a particular design for the application. We have also defined the nonfunctional requirements of the target system and developed the basic problem solving capabilities that are necessary in the PTA domain. In the next section, we will describe the environment in which the application will be located.

6.1.2 Environment View

The environment view of the agents within the IMRP domain is determined by the standards that are set by the FIPA. The FIPA Organization is an international consortium of about 50 industrial and academic institutions whose goal is to define standards for communication among agents to ensure interoperability in industrial applications. The FIPA97 (FIPA, 1997) specification provides technical standards for

- *Agent Management* (administration of agent systems, yellow page services, firewall technology etc.)
- *Agent Communication Language* (FIPA ACL based on speech-act theory, protocol specification etc.)
- *Agent Software Integration* (integration of existing SW systems in agent systems (agent wrappers) etc.)

The FIPA98 (FIPA, 1998) specification extends the FIPA97 documents by defining standards for

- *Agent Management Support for Mobility* (specification, configuration and handling of mobile agents)
- *Agent Security Management* (secure communication support, agent authentification, trusted platform specification)
- *Ontology Service Support* (support for the specification of the semantics of domain-specific message contents)
- *Human/Agent Interaction* (interface specification, user profiling, learning about preferences)

Since the PTA domain was created to evaluate the FIPA specifications, it is clear that the resulting application must conform to the standards described in (FIPA, 1997) and (FIPA, 1998).

The FIPA standard requires a minimal framework that provides primitive services for the agent inter-operation based on the following entities.

- The *Agent Management Service (AMS)* provides the services for the management and administration of agents eg. by offering a name service,
- the *Directory Facilitator (DF)* agent contains yellow pages with service descriptions and services and
- several *Agent Communication Channels (ACCs)* exist for platform-independent communication, secure communication and communication through firewalls.

The general development environment for the target system is based on three platforms (Solaris, Linux and NT) and uses Java (SUN MICROSYSTEMS, 1999) as the implementation language. The programming model is parallel because the individual route planer agents are supposed to be physically distributed and the agents will use network communication services to exchange messages.

6.1.3 Role View

6.1.3.1 Functional Roles

In the IMRP application, we can identify two functional roles. The *Unimodal Route Planner* is responsible for providing access to a particular travel resource and provides a planning service for this resource as defined below.

DEFINITION 6.1.9 (UNIMODAL ROUTE PLANER) *A function* $rp : V_M \times V_M \rightarrow R(M)$ *for a map* M *that returns a weighted route for any two nodes* $v_1, v_2 \in M$*, ie.* $rp(v_1, v_2) = ((v_1, \ldots, v_2), G)$*, is called a* **(unimodal) route planer** *for* M
Furthermore, the following conditions must hold:

- $(v_1, \ldots, v_2)$ *is a route from* v_1 *to* v_2 *and*
- G *is minimal.*

The second roles that occurs in the scenario is the *Intermodal Route Planner* that combines the services of various unimodal route planners in a single plan.

DEFINITION 6.1.10 (INTERMODAL ROUTE PLANER) *A function* $imrp : V_M \times V_M \rightarrow R(MAPS)$ *for a given set of maps is a* **intermodal route planer** *iff* $imrp$ *returns a list of unimodal maps* $((M, R, G)_1, \ldots, (M, R, G)_n)$ *for any two nodes* v_1 *and* v_2*, ie.* $imrp(v_1, v_2) = ((M, R, G)_1, \ldots, (M, R, G)_n)$.

6.1.3.2 Aggregation

The final design of the application uses two versions of role aggregation in order to evaluate the advantages and disadvantages of one approach or the other.

Broker In this version of the IMRP application, the functional roles are assigned to different agents, ie. there is one IMRP agent in the scenario that processes the user request and then tries to find UMRP agents that provide services that match the users requirements.

Distributed The distributed version of the application does not assign a fixed functional role to an agent, each agent can play both functional roles depending on the context in which it is incorporated into the joint planning process. If the agent is queried by the user, it acts as the IMRP the tries to integrate services provide by other agents with it own local service in order to generate a route plan. If, on the other hand, it is queried by another agent acting as IMRP, it offers its local service as normal UMRP.

6.1.4 Interaction View

The agent interaction that is necessary within the IMRP domain is limited to a rather simple requester-provider protocol. One agent will request a particular service from another agent which will serve the request and return the result to the requester. The application does not feature any market mechanisms or

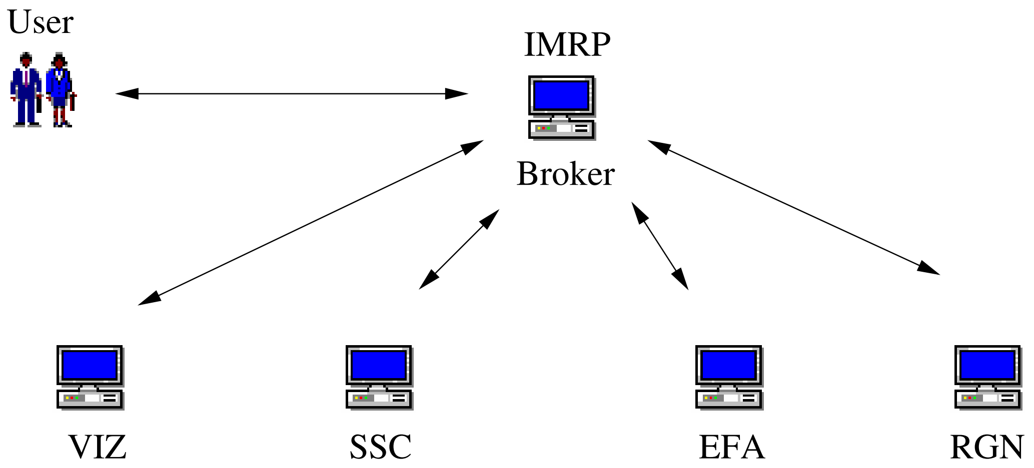


Figure 6.2: IMRP Broker Architecture

iterative refinements in the agent interaction process and is therefore straightforward to implement.

We have mentioned earlier that the IMRP application uses two different approaches to compute an intermodal route for a given routing scenario. In the following, we will describe, how the planning process works for each of these different approaches.

The control flow for the Broker version of the application is shown in Figure 6.2: the user request the a route planning service from the IMRP agent that in turn forwards partial requests to the UMRP agents and assembles the results according to the user preferences.

The computation uses a recursive algorithm based on the following idea:

- If the source and the destination node are in the same map or if the routing scenario consist of a single entry, the route can be computed by an individual route planner agent. Note that a routing scenario can be of length 1 only if the destination node is a junction point to the map of the source node.

- If the source and the destination node are in distinct maps, the optimal route (wrt. to a given optimization criterion) is the minimum of all routes between the source node and the intermodal route starting at any possible junction point to the next map.

The formal description of the algorithm is the following:

Input $v_s, v_d, rs = (M_1, \ldots, M_n)$

Algorithm

$imrp\_sc(v_s, v_d, rs)$ =

$$\begin{cases} rp_{Map(v_s)}(J_{M_{v_s}M_{v_d}}(v_s), v_d) & \begin{array}{c} Map(v_s) = Map(v_d) \\ or \quad |rs| = 1 \end{array} \\ \min\limits_{(j_1,j_2)\in J(M_1,M_2)} \begin{array}{l} append(rp_{M_1}(v_s, j_1), \\ imrp\_sc(j_2, v_d, tail(rs))) \end{array} & otherwise \end{cases}$$

Require: $rs \in RS(v_s, v_z)$

```
if Map(v_s) == Map(v_d)||(|rs| == 1) then
    minroute := rp_{Map(v_s)}(J_{M_{v_s} M_{v_d}}(v_s), v_d)
else
    minroute := ∅
    for all (j_1, j_2) ∈ J(M_1, M_2) do
        route := rp_{M_1}(v_s, j_1) + imrp_sc(j_2, v_d, tail(rs))
        if Γ(route) < Γ(minroute) then
            minroute := route
        end if
    end for
end if
return minroute
```

Output The optimal path between $v_s v_d$ for a fixed routing scenario rs wrt. to the weight functions γ, Γ.

In the distributed version of the application, the control flow in the application is as shown in Figure 6.3: The user selects the most appropriate UMRP agent (eg. the service wrapper for the first service that the user intends to travel with) and places a request. The UMRP will then play the role of the IMRP in subsequent service requests to other UMRP agents and return the result to the user.

The algorithm that implements this flow of control is given below:

Input v_s, v_z, rs

Algorithm

$imrp\_sc(v_s, v_z, rs) =$

$$\begin{cases} rp(v_s, v_z) & Map(v_s) = Map(v_z) \\ \min\limits_{(j_1,j_2)\in J(M_1,M_2)} \begin{array}{l} append(rp(v_s, j_1), \\ RP_{M_2}(j_2, v_z, tail(rs))) \end{array} & otherwise \end{cases}$$

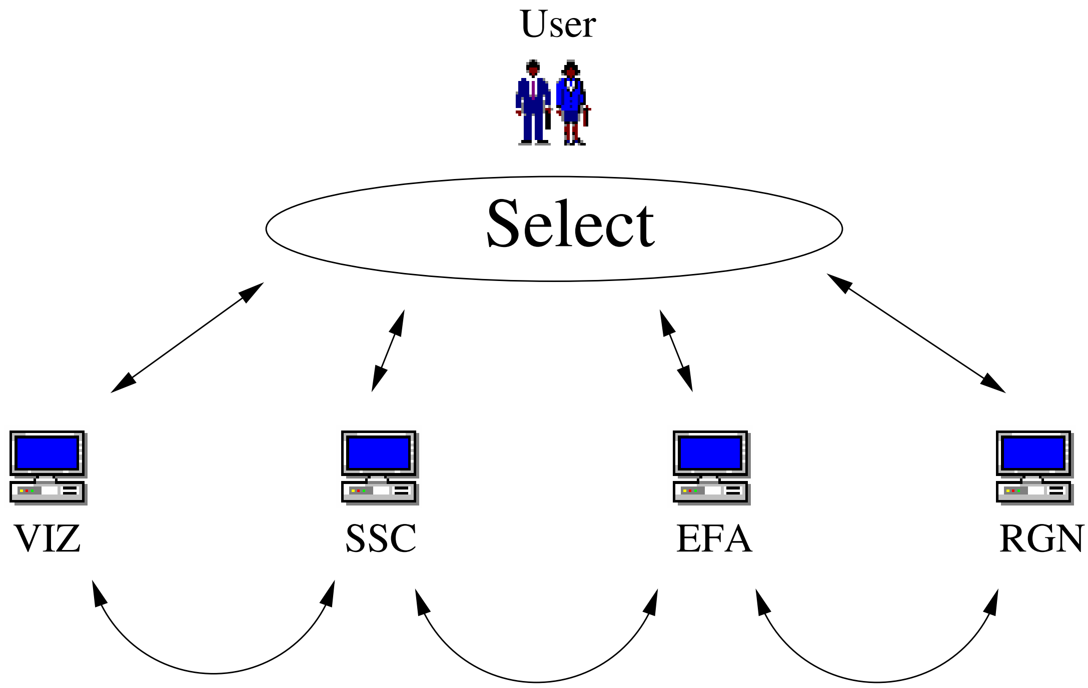


Figure 6.3: Distributed IMRP Architecture

if $Map(v_s) == Map(v_d)||(|rs| == 1)$ **then**
    $minroute := rp_{Map(v_s)}(J_{M_{v_s}M_{v_d}}(v_s), v_d)$
else
    $minroute := \emptyset$
    for all $(j_1, j_2) \in J(M_1, M_2)$ **do**
        $route := rp_{M_1}(v_s, j_1) + RP_{M_2}(j_2, v_z, tail(RS))$
        if $\Gamma(route) < \Gamma(minroute)$ **then**
            $minroute := route$
        end if
    end for
end if
return $minroute$

Output The optimal path between $v_s v_d$ for a fixed routing scenario rs wrt. to the weight functions γ, Γ.

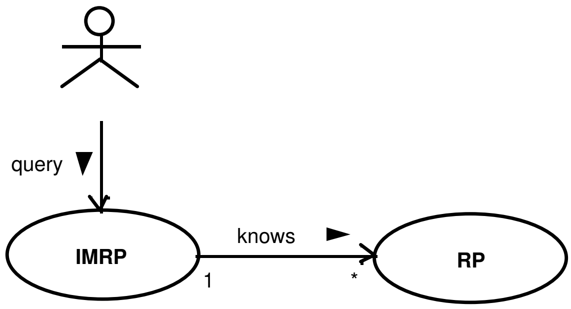


Figure 6.4: Basic Society Structure

6.1.5 Society View

The society that operates in the IMRP domain is rather simple and consists of two or three different entities, depending on which of the alternatives (broker or distributed version) is considered. In either case, however, the structure of the society is initially flat as shown in Figure 6.4. The user connects to the IMRP agent which in turn calculates an intermodal route for the user request by connecting the individual travel services and then returns the intermodal travel plan to the user.

This simple form of the society structure is adequate as long as the number of individual travel services is relatively small. However, in a scenario with a large amount of individual travel service, eg. when each UMRP represents some local public transport facility, this society structure may no longer be adequate. The IMRP agent cannot compute an intermodal travel plan in a one-shot process but it must connect some intermediate IMRP agents that interface to the local travel services. The respective society structure as shown in Figure 6.5 results in a control flow that operates at several hierarchy levels. First of all, the user queries the local IMRP agent that is responsible for the area the user is currently located. Next, this IMRP agent forwards part of the user request to other local IMRP agents that compute a partial intermodal route for their area and return the solution to the first IMRP agent that assembles a full intermodal route for the user request and returns the solution to the user.

In this section, we have modeled the society structure of the IMRP application and outlined some extension to the structure as it will be implemented in a the first version of the system. In the next section, we will illustrate how the various features that were designed in the previous views are transformed in a coherent system architecture.

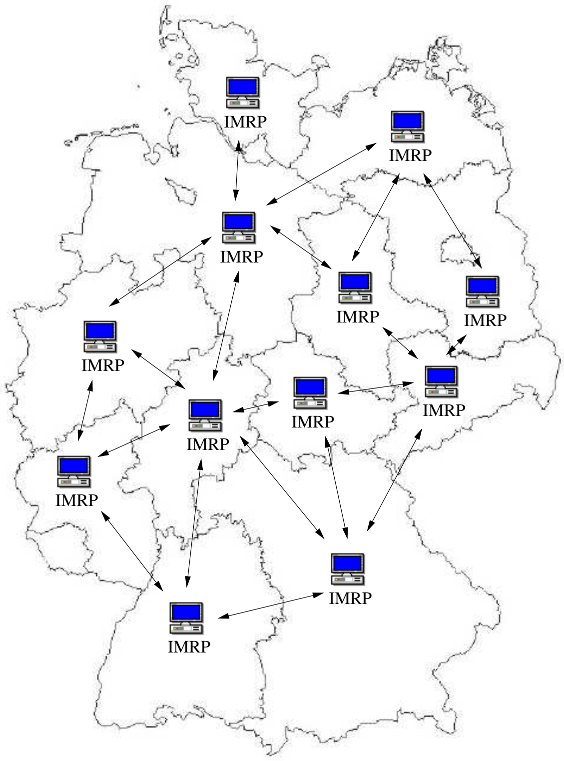


Figure 6.5: Hierarchical IMRP Organization

6.1.6 Architectural View

6.1.6.1 System Architecture

In this section, we will characterize the required system architecture according to the scheme given in Section 5.7.1.

Entities The only entities that occur in the target system are agents for unimodal route planning and intermodal route planning, respectively.

Control flow Because of the fact that the only entities are agents, the control flow within the system is determined by the Interaction view. The same applies for the information flow.

Agent management Since the problem domain was chosen as an evaluation domain for FIPA standards, the agent management will be handled according to the requirements specified there.

Communication model The system architecture should allow for a geographical distribution of the system components and therefore provide some network based message passing mechanisms. In order to be able to debug the application locally, it is desirable to have local message passing that can be traced within a single debugger.

Agent architecture The agent architecture will be discussed in the next section.

Database design Since the individual agents within the system do not share data explicitly, it is not necessary to provide a system wide database.

External components/devices The external components that must be integrated in the target system are the existing travel services that are run by external providers. These services must be encapsulated in agents in order to achieve the required degree of inter-operability.

Thus, the overall system architecture is determined by the concepts specified in the FIPA standard that is used as the general basis of the application. According to the above requirements, we have chosen the MECCA agent application framework (STEINER, 1992) as the implementation basis for the actual system because MECCA is a FIPA compliant framework that provides libraries of pre-defined agents according to the FIPA standard and includes a generic agent architecture as well as a protocol specification language for the agents.

The target system is implemented in a fully parallel system environment where all agents are running on different host computers. However, the difficult task of debugging a fully distributed system is eased as the MECCA framework offers a transparent scheme for running the agents locally without any changes to the code.

6.1.6.2 Agent Architecture

According to the characterization given in Section 5.7.1, the agent requirements are as follows.

Reasoning capabilities According to their actual role, the agents need some planning facility that enables them to find an unimodal or an intermodal route using the given travel service.

Resources limitations The resource requirements of the agents is low because the planning process mentioned above is not highly complex. However, due to the necessary encapsulation of external services, the agents will be equipped with the computational resources of a complete workstation.

Control flow The control flow within the agent is determined by the algorithms described in Section 6.1.4 and is basically a sequential one. It is therefore not necessary for the agent architecture to provide a parallel execution model.

Knowledge Representation The major knowledge structures used by the agents are the unimodal and intermodal plans. These plans must be explicitly represented in a knowledge base as they are likely to change in the course of the route planning process and because they must be transfered to a human-readable form.

Autonomy The agents in this limited part of the full scenario do not exhibit any pro-active behavior as the only react to a user request or to a request from another agent to provide some well-defined service. In the full scenario, however, the agent are supposed to become active whenever the external state as observed by the agent contains some information that may be useful for their user. Still, this is not subject of this part of the system.

User interaction The degree of user interaction is not very high for this part of the system and is limited to the route specification in form of start and destination nodes, travel times, preferred travel modalities and probably a few other things. Therefore, the agents user interface will not need any particular attention.

Temporal context The lifetime of an agent in the target system can be quite long as the unimodal route planners are supposed to encapsulate real-world services that are available 24 hours a day. However, the agents do not need any persistence mechanisms because these are already present in the underlying services.

Rationality

Because of the above requirements, we have chosen the MECCA agent architecture that was specifically designed to serve as wrapper for proprietary software systems that are to be used in a multiagent systems and are thus well-suited for the encapsulating of the individual travel services. A MECCA agent consists of three parts as shown in Figure 6.6.

The agent *body* encapsulates application specific functionalities eg. existing SW systems, databases etc., the *head*: contains the goals and plans of the agent and controls the body via the application interface. Furthermore, it also contains the protocol execution unit that can manage several contexts at the same time. The *communicator*, finally, is the physical communication interface that implements low-level communication protocols such as TCP/IP, GSM or IIOP.

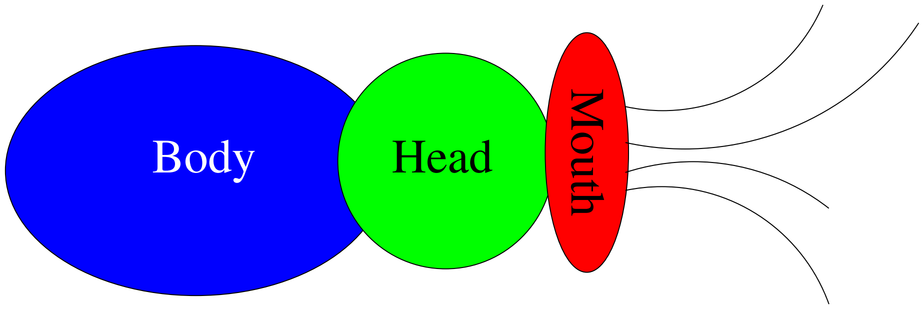


Figure 6.6: MECCA Agent Architecture

6.1.7 System View

6.1.7.1 Graphical User Interface

The User interface of the IMRP system as shown in Figure 6.7 consists of five major elements that are briefly described below.

Map Visualizer The Map visualizer window contains the graphical representation of the unimodal maps that are managed by the unimodal route planners. The user selects the start and the destination node of the journey and is then queried for the temporal constraints. The system will then calculate an intermodal route according to the provides specification.

Test Configuration Tool In order to evaluate the different approaches presented earlier, the user can select between the broker and the distributed variant and between the optimal computation of the intermodal route and the use of a heuristic as is described in Section 6.1.7.2.

Heuristic Configuration Tool This window allows the user to select the heuristic and the parameters of the heuristic. The heuristics that can be used as well as their parameter settings are described in Section 6.1.7.2.

Status Window The Status window informs the user about the result of the computation (eg. the length or the duration of the intermodal route) and about exceptional situations such as constraint inconsistencies that occur eg. when the time window is to tight for particular start and destination nodes.

Agent Directory Service This window, finally, is provided by the FIPA compliant MECCA agent platform and contains the names of the agents currently

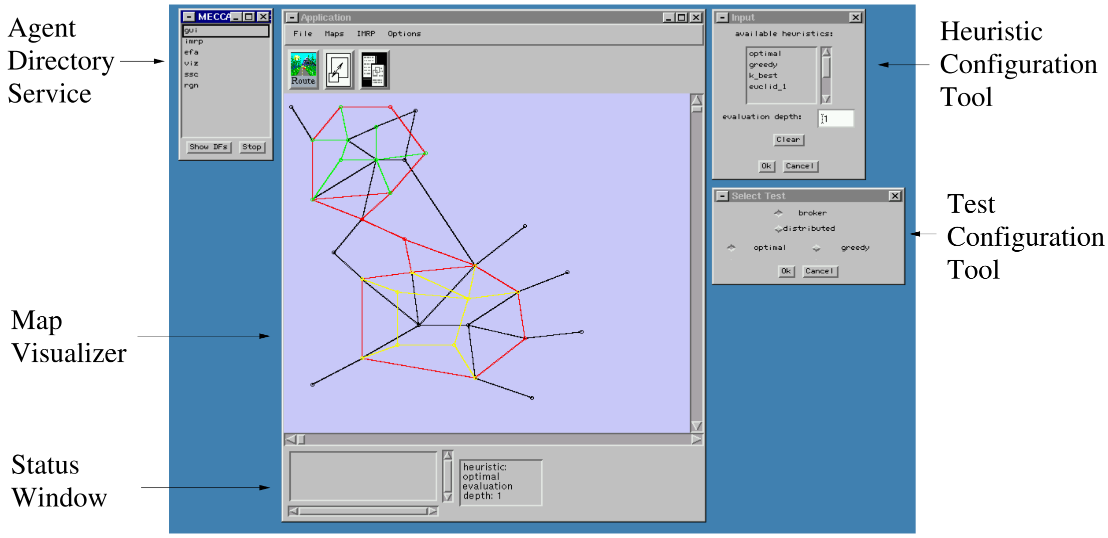


Figure 6.7: IMRP Graphical User Interface

known to the system. The user can click on the agent name to obtain additional information about the respective agent.

Since the intermodal route planning facility is only a service that is provided in a larger context, the user interface is only very fragmentary. It is, however, functional enough to enable the user to experiment with the different variants of the the system. Experiments that were conducted with early versions of the system showed, that the system performance might become unacceptable if the number of unimodal maps and the size of these maps increases beyond a particular point. Therefore, several attempts were made to obtain rather good – though not optimal – solutions with a reduced effort. The results of these attempts are discussed in the following section.

6.1.7.2 Performance Engineering

Suboptimal Solutions Until now, the IMRP agent has computed optimal routes for each user request which implied the exploration of the entire search space. In this section, we present some search heuristics that aim at finding sufficiently good solutions with less computational effort. The main idea of the search heuristics discussed below is a flexible steering of the search process by narrowing the set of junction points between two adjacent maps in a routing scenario.

Therefore, we define a selection function for the junction points that should be expanded during the search process as follows.

DEFINITION 6.1.11 (SELECTION FUNCTION) *Define* $select : (V, V)^n \rightarrow (V, V)^m, n \leq m$ *as*

$$select(\{v_i, v_j | (v_i, v_j) \in J(M_k, M_{k+1})\}) = \{v_o, v_p | (v_o, v_p) \in J(M_k, M_{k+1})\}$$

ie. $select$ *chooses an arbitrary subset of the junction points for any two maps.*

Now, by using the $select$ function from definition 6.1.11, we extend the search algorithm such that it expands only the selected nodes. We demonstrate how this extension is implemented using the broker architecture discussed earlier, the changes apply in an analogous way to the distributed architecture. Now, the modified broker algorithms is given below.

Input $v_s, v_z, rs = (M_1, \ldots, M_n)$

Algorithm

$imrp\_sc(v_s, v_z, rs) =$

$$\begin{cases} rp_{Map(v_s)}(J_{M_{v_s} M_{v_z}}(v_s), v_z) & \begin{array}{c} Map(v_s) = Map(v_z) \\ oder \quad |rs| = 1 \end{array} \\ \min\limits_{\substack{(j_1, j_2) \in \\ select(J(M_1, M_2))}} \begin{array}{l} append(rp_{M_1}(v_s, j_1), \\ imrp\_sc(j_2, v_z, tail(rs))) \end{array} & otherwise \end{cases} \quad (6.1)$$

Require: $rs \in RS(v_s, v_z)$

if $Map(v_s) == Map(v_d) || (|rs| == 1)$ **then**
  $minroute := rp_{Map(v_s)}(J_{M_{v_s} M_{v_d}}(v_s), v_d)$
else
  $minroute := \emptyset$
  for all $(j_1, j_2) \in select(J(M_1, M_2))$ **do**
    $route := rp_{M_1}(v_s, j_1) + imrp\_sc(j_2, v_d, tail(rs))$
    if $weight(route) < weight(minroute)$ **then**
      $minroute := route$
    end if
  end for
end if
return $minroute$

Output A path between $v_s v_d$ for a fixed routing scenario rs

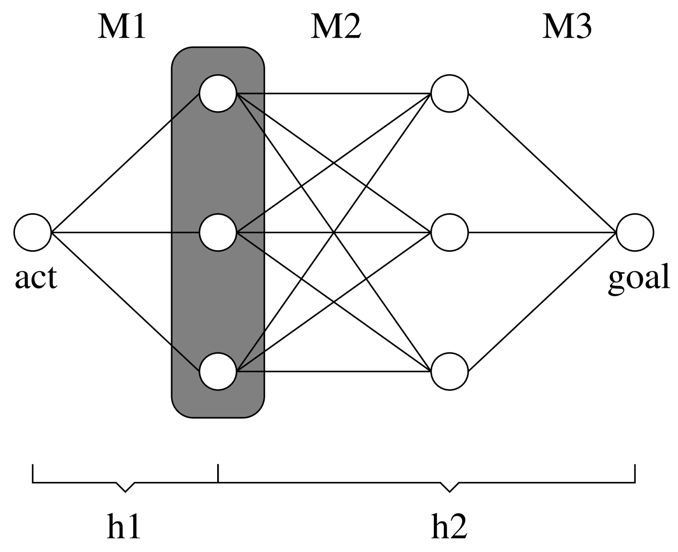


Figure 6.8: Generic Selection Function

As we have seen, the integration of the selection function is quite straightforward but it is yet unspecified how a particular selection might work to produce satisfactory results. Therefore, we first of all define a *generic selection function* and then instantiate this generic function to obtain specific selection functions.

The generic $select$ function operates in three steps:

(1) Generate a partial ordering on the set of possible junction points.

(2) Sort the junction points according to the partial ordering.

(3) Select the best candidates.

In Figure 6.8, we have illustrated the process. In the first step, we assign a weight to each junction point x with respect to the current search node act and the destination node $gaol$ according to

$$g(x) = h_1(act, x) + h_2(x, goal) \tag{6.2}$$

where h_1 estimates the weight of the route from the current node to the junction point and h_2 estimates the weight of the route between the junction point and the goal. Now we can sort the list according to the combined weights and select the best nodes according to a given evaluation depth.

In the following description, we discuss some instantiations of the generic $select$ function by specifying the two estimation functions h_1 and h_2 and the evaluation depth of the final selection step.

Optimal Selection This instantiation realizes the standard case without narrowing the search space. All possible junction points are selected which guarantees, that the optimal solution is found. We have included this instantiation in the description for the sake of completeness.

$$h_1(act, x) = weight(rp(act, x))$$
$$h_2(x, goal) = weight(rp(x, goal))$$

Obviously, all junction points must be selected in order to guarantee optimality in the search process.

Greedy Search This variant of the $select$ function considers only the locally best alternatives. Therefore, we first define the following concepts.

DEFINITION 6.1.12 (K-BEST SEARCH) **k-best search** *expands only the first* k *junction points that have the shortest distance to the source node of the route.*

DEFINITION 6.1.13 (K-BEST JUNCTION POINTS) $topJPs : UMAPS \times UMAPS \times V \times \mathbb{N} \mapsto (V, V)^n$ *is defined as*

$$topJps(M_1, M_2, v, k) = ((j_{11}, j_{12}), \ldots, (j_{k1}, j_{k2})) \textit{ with}$$
$$(\forall i \in \{1, \ldots, k\} : (j_{i1}, j_{i2} \in J(M_1, M_2) \wedge$$
$$(\exists l \in L_M, i \in \{1, \ldots, k\} : \alpha(l) = v \wedge \in \omega(l) = j_{i1}) \wedge$$
$$(\forall j \notin j_{il} \forall i \in \{1, \ldots, k\} : \gamma(v, j) > \gamma(v, j_{i1})))$$

We obtain a greedy algorithm by setting
$select(J(M_1, M_2)) = topJPs(M_1, M_2, v_s, 1)$. This corresponds to the following setting

$$h_1(act, x) = weight(rp(act, x))$$
$$h_2(x, goal) = 0$$

In the second step of the selection process, only the first junction point in the ordered set is expanded, an obvious extension of the algorithm is to select not only the best junction point, but the k best junction points by setting $select(J(M_1, M_2)) = topJPs(M_1, M_2, v_s, k)$ for some fixed value of k.

Euclidean Distance Measure The Euclidean distance between two points $p = (x_1, y_1)$ and $q = (x_2, y_2)$ is defined as

$$d(p,q) = d((x_1,y_1),(x_2,y_2)) = \sqrt{(x_1-x_2)^2 + (y_1-y_2)^2} \tag{6.3}$$

From this definition, we can derive three different heuristics:

$$\begin{aligned} h_1(act,x) &= weight(rp(act,x)) \\ h_2(x,goal) &= d(x,goal) \end{aligned}$$

$$\begin{aligned} h_1(act,x) &= d(act,x) \\ h_2(x,goal) &= weight(rp(x,goal)) \end{aligned}$$

$$\begin{aligned} h_1(act,x) &= d(act,x) \\ h_2(x,goal) &= d(x,goal) \end{aligned}$$

The k best junction points are selected from the ordered set of junction points according to the given evaluation depth.

Precedence Ordering on Maps This heuristic uses a partial ordering defined on the set of unimodal maps. We define a weight function $w : UMAPS \rightarrow \mathbb{R}$ that assigns a weight to each unimodal map. We set

$$\begin{aligned} h_1(act,x) &= weight(rp(act,x)) * w(Map(act)) \\ h_2(x,goal) &= weight(rp(x,goal)) * w(Map(x)) \end{aligned}$$

to obtain the map precedence heuristic.

Figure 6.9 shows an example with two maps indicated by the solid and the dashed line, respectively. The solid line map represents the public transport service of a city, whereas the dashed map represents the interregional individual traffic.

Using this heuristic, a route query from "source" to "destination" will yield a solution as it is shown in Figure 6.10. This solution reflects the shortest path from the source to the destination node, but it neglects the fact that the public transport is usually much more time consuming then individual traffic. Generally speaking, it is often more efficient to leave the city as fast as possible and to use high speed individual traffic routes around the city.

Using a precedence ordering on maps (and implicitly on means of transport), the higher preference for individual transport towards public transport yields the result presented in Figure 6.11.

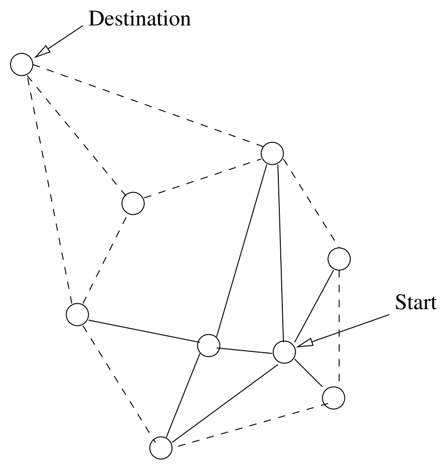


Figure 6.9: Example 1

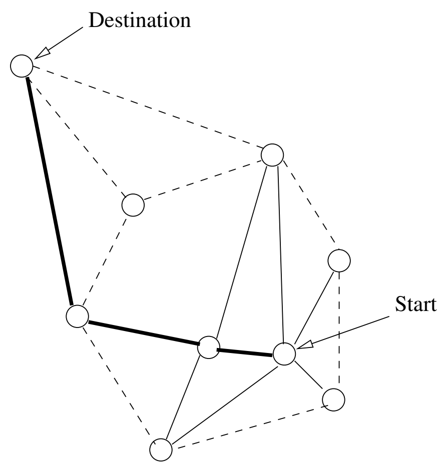


Figure 6.10: Example 2

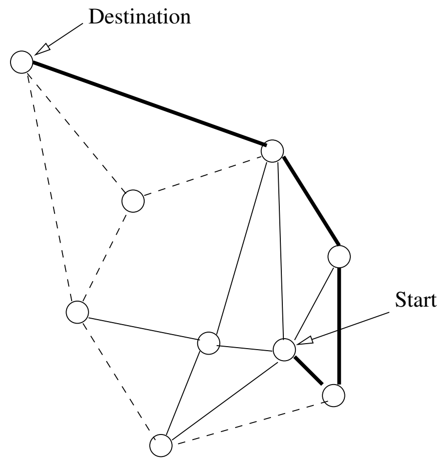


Figure 6.11: Example 3

Caching The maximum number of unimodal routes that must be computed for an intermodal route $(U_1, \ldots, U_n)$ is a function of the number of junction points between the unimodal maps for a particular routing scenario. For a given routing scenario $rs = (M_1, \ldots, M_m)$, the number of possible unimodal routes ($\#_{rs}$) computes as follows:

$$\begin{aligned}
\#_{rs}(v_s, v_z, ()) &= 0 \\
\#_{rs}(v_s, v_z, (M)) &= 1 \\
\#_{rs}(v_s, v_z, (M_1, \ldots, M_m)) &= |J(M_1, M_2)| + \\
&\quad \sum_{(j_1, j_2) \in J(M_1, M_2)} \#_{rs}(j_2, v_z, (M_2, \ldots, M_m))
\end{aligned} \tag{6.4}$$

The recursive formula (6.4) is not optimal for practical purposes and therefore we also give the number of possible unimodal routes in a non-recursive form. Note that we define $|J(M_m, M_{m+1})| = |J(M_{m+1}, M_{m+2})| = 1$ in Formula 6.5.

$$\#_{rs}(v_s, v_z, rs) = \sum_{i=1}^{m+1} \prod_{j=1}^{j \leq i} |J(M_j, M_{j+1})| \tag{6.5}$$

Comparing this to the number of *unique* unimodal routes which computes to

$$\#_{unique}(v_s, v_z, rs) = \sum_{i=1}^{m-1} |J(M_i, M_{i+1})| * |J(M_{i+1}, M_{i+2})| \tag{6.6}$$

reveals a high potential for performance gains if the re-computation of unimodal routes can be avoided. For example, given three maps with 10 junctions points between each of them, we obtain a total of 111110 unimodal routes that must be combined according to equation 6.5 out of which only 300 are unique according to equation 6.6. To take profit from this discrepancy, we define a generic cache as follows.

DEFINITION 6.1.14 (GENERIC CACHE) *A* **cache** *is a data structure that stores* $(key, value)$ *pairs where* $key \in KEY$ *and* $value \in VALUE$ *We define functions* $store, retrieve$ *and* $flush$ *that operate on these* $(key, value)$ *pairs.*

$store : KEY \times VALUE \rightarrow \{success, failure\}$

$$store(k, v) = success, C' = (C \backslash \{(k, v') | v' \neq v\}) \cup \{(k, v)\} \tag{6.7}$$

$retrieve : KEY \rightarrow VALUE$

$$retrieve(k) = \begin{cases} v & falls(k, v) \in C \\ \emptyset & sonst \end{cases} \tag{6.8}$$

$flush : \rightarrow \{success, failure\}$

$$flush() = success, C' = \emptyset \tag{6.9}$$

Now, whenever the IMRP needs a particular unimodal route, it first checks the cache whether it already contains the route or not. In the first case, the route is used directly whereas in the second case, the IMRP agent issues a request to the UMRP in charge and then stores the result in the cache for potential reuse.

6.1.8 Summary

In this section, we have demonstrated, how the MASSIVE method was used in the trip planning domain. In the next section, we will present another case study where MASSIVE was used to build a generic software library.

6.2 The TEAMWORK LIBRARY

The TEAMWORK LIBRARY (LIND, 1996A), (DENZINGER AND LIND, 1996) is a framework for the development of distributed search applications according to the Teamwork approach (AVENHAUS AND DENZINGER, 1993), (DENZINGER, 1993). In the following sections, we present the teamwork idea and the design of a library that supports the user in developing teamwork-based applications.

6.2.1 Task View

In this section, we present the key ideas of the teamwork approach as they were introduced in (DENZINGER, 1993) in the context of equational theorem proving and later extended and generalized in (DENZINGER AND LIND, 1996).

Functional Requirements The teamwork approach is a distributed problem solving algorithm for problems with very large (or infinite) search spaces. Due to the size of the problems of interest, it is impossible to traverse the entire search space, and in some cases (e. g. optimization problems), a rather good, although not the best, solution is usually sufficient. Since there is a large number of possibilities to proceed with the search it is very important to design heuristics or knowledge based algorithms to concentrate on search paths which will lead to the goal quickly.

We can identify two groups of algorithms that have their main difference in the information they need to continue the search process. The first group are search processes that need explicit information about prior search states, i. e. they need to know the history of the search. The processes of the second group do not need any history information to proceed with their search.

The first group of search processes are called *divide and conquer* algorithms and they normally use trees or directed graphs to represent their states. Examples for search processes of this group are the well known Branch and Bound algorithms (LAWLER AND WOODS, 1966). The way they execute their search is quite straightforward: the problem is divided into several subtasks which are worked on independently, i.e. without any communication between the tasks. The results of the subtasks are assembled at the next higher level and, since the search may lead to dead ends, backtracking is sometimes necessary.

The main problem for this group of search processes is to design good heuristics to identify the subtasks into which the main task will be divided. Load balancing between the physical agents to which the subtasks are assigned and identifying subtasks that can actually be solved by a single agent are key issues for the design of an application that uses these search processes.

In the second group, we have algorithms that perform a *search by extension and focus.* In this group of search processes, the search space is represented by a set of results which is normally unstructured. The search processes do not keep explicit information on how a search state was reached, i. e. no information about the decisions which lead to a particular state are kept. To determine the transitions from one search state to another, the search processes use a set of *extension* rules. Due to the problem size, there are usually a very large number of possible extensions and this is why a *focus* function is needed to limit this number. A characteristic property of problems that can be solved with algorithms of this group is that it is normally not possible to find "natural" partitions of the search space. This is the reason why the teamwork approach was developed for this class of search processes since divide and conquer algorithms are not suitable for these kinds of problems.

The distinction of problems into two classes is not very sharp: There are problems that may be solved by using algorithms from either group, for example to prove the unsatisfiability of a formula of first order predicate logic. This problem can be solved with a semantic tableau (SMULLYAN, 1968) which belongs to the first group, or by using resolution and factorization (ROBINSON, 1965) which belongs to the second group. The distinction is, however, precise enough to characterize the problems for which the teamwork approach can be used.

In order to formalize the concepts that have been introduced so far, we have the following definitions.

DEFINITION 6.2.1 (SEARCH BY EXTENSION AND FOCUS) Search by extension and focus *is described by a 4-tuple* $(\mathcal{B}, \Omega, \mathcal{I}, \mathcal{S}_\prime)$. *Here* $\mathcal{B}$ *is a set of objects and the subsets* $S \in 2^{\mathcal{B}}$ *of* $\mathcal{B}$ *are called* states. Ω *is a predicate defined on* $\mathcal{B}$ *and* $\mathcal{I}$ *consists of* extension *rules of the form* $L \cup M \to L \cup N$ *with* $L,M,N \in 2^{\mathcal{B}}$. $S_0 \in 2^{\mathcal{B}}$ *is called the* start state *and for all* $s \in S_0$ *we require that* $\Omega(s)$ *holds. We write* $S \vdash_{\mathcal{I}} S'$ *for states S and S', if there is a rule* $L \cup M \to L \cup N$ *in* $\mathcal{I}$ *such that* $S = S'' \cup L \cup M$ *and* $S' = S'' \cup L \cup N$ *and* $\Omega(s)$ *holds for all* $s \in S'$. *A sequence* $(S_0, S_1, \ldots, S_n)$ *with* $S_{i-1} \vdash_{\mathcal{I}} S_i$ *for all i=1,...,n is called a* search derivation. *We call the* S_i derivable *from* S_0.

$\mathcal{B}$ and $\mathcal{I}$ are determined by the (general) problem to be solved and $\mathcal{I}$ is (since it may contain infinitely many elements) usually represented by a finite set of rule schemata. Note that Ω and S_0 are determined by the instance of a problem, i. e. the input of the search process.

DEFINITION 6.2.2 (GOAL OF A SEARCH BY EXTENSION AND FOCUS) *Let* $(\mathcal{B}, \Omega, \mathcal{I}, \mathcal{S}_\prime)$ *be a search by extension and focus. Let* $>_{\mathcal{B}}$ *be a Noetherian partial ordering on* $\mathcal{B}$. *Then we call an element* $g \in \mathcal{B}$ *having property* Ω *that is* *minimal with respect to* $>_{\mathcal{B}}$ *a* goal *of the search. A state containing g is called a* goal state. *The goal is* reachable*, if there is a goal state* S_i *derivable from* S_0 *by rules from* $\mathcal{I}$.

The definitions given above are not sufficient to completely define a search process because some indeterminism remain unresolved. It is, for example, possible that the goal cannot be reached for a given input or there may be several states containing the goal. These problems are solved by providing a function to select the "best" extension for a given search state. This function is called the *focus* of the search process.

DEFINITION 6.2.3 (FOCUS FUNCTION) *Let* $(\mathcal{B}, \Omega, \mathcal{I}, \mathcal{S}_/)$ *be a search by extension and focus and g a goal. A function f:*$2^{\mathcal{B}} \times \mathcal{I} \to Z$ *is called a* focus function *and the derivation* $(S_0, S_1, ..., S_i, ...)$ *is produced by f, if for the extension* $L_i \cup M_i \to L_i \cup N_i$ *that produced state* S_i*,* $i > 0$*, we have that* $f(S_{i-1}, L_i \cup M_i \to L_i \cup N_i) \leq f(S_{i-1}, L \cup M \to L \cup N)$ *for all* $L \cup M \to L \cup N \in \mathcal{I}$.

Nonfunctional Requirements The nonfunctional requirements of the TEAMWORK LIBRARY are as follows. First of all, the library should be *easy to use* even for inexperienced users. Thus, the library should provide the necessary structures for a broad range of applications that can be used with little adaptations by the application developer. Secondly, the library should be *flexible and generic* in a way that it supports different application classes on a high level of abstraction and provide the mechanisms to instantiate the generic base structures for a concrete problem. Finally, the library should be *efficient* in that is has only little administrative overhead and efficient algorithms for aspects such as data transfer etc.

Basic Problem Solving Capabilities The basic problem solving capabilities are domain specific aspects that are not relevant to the development of the TEAMWORK LIBRARY just because it is a library.

6.2.2 Environment View

Since the TEAMWORK LIBRARY is intended as a generic framework for distributed search application in various contexts, it is not possible to describe the environment from the perspective of the agents because this view depends on the particular problem domain. Therefore, we are limited to the developers perspective in the description of then Environment view of the TEAMWORK LIBRARY.

This perspective is dominated by the characteristics of the runtime environment which in turn is determined by the intended initial use of the library in the context of a Solaris environment. Due to efficiency aspects, the goal language was chosen to be C++ (STROUSTRUP, 1987) and the programming model should be parallel because a pseudo-parallel approach would not adequately exploit the benefits of the teamwork approach. Finally, the communication platform provided by Solaris is TCS/IP.

Especially the last point had some major impact on the overall structure of the TEAMWORK LIBRARY and will be discussed in Section 6.2.4.

6.2.3 Role View

6.2.3.1 Functional Grouping

The general idea of the teamwork approach is to work with different algorithms on the same problem and to incorporate the specific benefits of each algorithm into an overall solution that is better than any of the separate solutions alone. This will not work in all cases, but experience shows, that it can – and in most cases will – lead to better results than a single algorithm alone (PITZ, 1993).

The original teamwork approach as presented in (DENZINGER, 1993) was inspired by modeling a human project team on a computer network. In bigger companies, it is quite common to have project teams working on a specific problem by using the knowledge and experience of several experts. There is a large number of different ways to organize project teams and the teamwork approach is is just one way how to organize it, but since it is not possible and not ingenious to try and cover all possibilities, some assumptions are made to concentrate on one possible realization. In the teamwork view, the composition of a team (i. e. the team *members*) can be adjusted to a given problem and the composition can change during the problem solving process. The control of the whole process lies within a (team-)supervisor on top of the team hierarchy. It is, however, possible to delegate complete (sub-)tasks to a subteam, but the main responsibility for the problem solving process stays with the supervisor of the main team. The teamwork approach tries to reflect such a human project team that consists of four types of components: *experts, specialists, referees* and a *supervisor*.

The teamwork approach distinguishes two phases of the problem solving process. During the *working phase*, the problem solving entities generate search states according to their individual specifications. Each of these working phases is followed by a *team meeting* during which the more administrative aspects of the problem solving process are dealt with.

Experts and *Specialists* are the only components of a team which work on the problem directly. Each of the different experts must be capable to solve the problem alone, but, because of the different algorithms and heuristics used in the experts, each of the experts has another view on the problem. A very important need for an expert is that it must be able to continue the work started by another expert. This implies, that in contrast to specialists as we will see later, all experts in a team must have the same knowledge representation. Another restriction for the implementation of an expert is the size (the duration) of an atomic step in its computation. This size must not be too large because an expert must be able to stop its computation in a stable state at almost any time to join a team meeting. It is not acceptable for other experts to waste time on waiting for experts to be ready for a team meeting.

Specialist differ from experts mainly in two ways:

- they may have their own data representation and
- they may keep knowledge for a longer time (especially for more than one team meeting)

The second point will be explained later, when team meetings are discussed.

An important requirement for specialists is that they must be able to convert their results (represented in their special knowledge representation) into a format that can be understood by the experts.

The *referees* mentioned above have two major tasks:

- assessing the work of an expert or a specialist to judge the progress made
- choosing "good" results computed by an expert/specialist

The main difference among the referees is their assessment strategy, which, in most cases, are statistical criteria. The assignment of referees to the experts/specialists they assess is not fixed during the entire application. Instead, it is possible that an expert/specialist is assessed by different referees during a system run. This reflects the fact that the problem description evolves during the solution process and that the assessment criteria for what is regarded as a "good" result or a "big" progress may change. Note that a special knowledge representation of the specialists requires special referees to assess the specialists work and to choose the good results.

The reports generated by the referees are used to determine the next team supervisor and the results chosen to be good offer the possibility to forget unimportant facts which did not contribute to the solution process. This forgetting of facts is very important in order to avoid blowing up the search space.

The *supervisor* of a team, finally, has the central control over the team activities. Its main tasks are:

- to build a new, improved problem description before the next working phase starts,
- to choose the experts to work in the next working phase, and
- to determine the duration of the next working phase

6.2.3.2 Physical Grouping

The physical grouping of the roles in the Teamwork application is determined by the assumed complexity of the problem solving capabilities of the agents. In the first application of the teamwork approach (PITZ, 1993), each agent was a full fleshed theorem prover that needed the computational resources comparable to those of a Unix process. The teamwork approach was originally designed for coarse-grained problems that are worked on by resource-intensive inference mechanisms and this fact is reflected in the physical grouping of the agents as well. Thus, as in the very first Teamwork application, we assume that each agent is equipped by the amount of resources mentioned before.

6.2.3.3 Aggregation

The functional and the physical physical groupings that were presented in the previous sections map together very well making it easy to define each role according to the functional grouping because the physical grouping is well capable of satisfying the resource requirements of the functional grouping.

Now that the roles within the TEAMWORK LIBRARY have been defined, it is the next step to specify the interactions that take place between these roles.

6.2.4 Interaction View

The overall control flow of a teamwork application consists of repeated loops of working phases until either a solution is found or the search process is aborted. At the end of each working phase, a team meeting is held to bring together the results of the experts and specialized that have worked on the problem for some time, to evaluate and combine their respective results and to select the next team supervisor and the participants of the next work cycle.

Figure 6.12 shows the information and control flow during a team meeting. During a working phase, the team supervisor acts as a normal expert, its special task starts with the beginning of the meeting. First of all, the supervisor

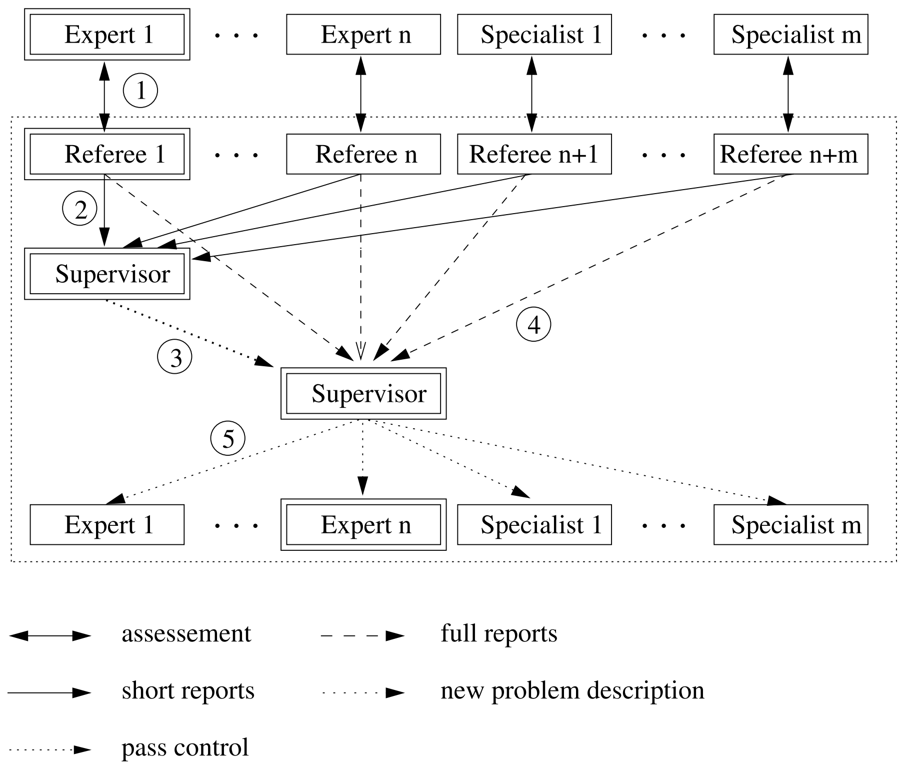


Figure 6.12: Team Meeting

must determine the best expert which will later become the next supervisor. Therefore, the referees start to evaluate the results that were obtained by the experts and specialist during the last working phase (①) and the current supervisor uses short reports (②) (usually containing only a single assessment value) of the referees to select its successor. Control is then instantly passed to the new supervisor (③) that takes over the lead of the team meeting started by the old supervisor.

Firstly, the new supervisor requests the detailed reports from the referees (④). Each report contains the good results and several statistical data, generated on the performance evaluation of the experts or specialists. After receiving all data, the supervisor starts to integrate the results of the experts and specialists into its problem description. Note that this is the problem description of the best expert during the last working phase and thus ideally contains a problem description that is more likely to contain the solution then the descriptions of any other agent. Then, the statistical data of the reports is used to choose the team members for the next working phase by replacing unsuccessful experts by others which are expected to provide better results. This selection of experts and specialists enables the supervisor to adapt the team configuration to the given problem.

In the final step of the team meeting, the supervisor determines the duration of the next working phase and transfers all necessary knowledge (problem description, team structure and duration of the next working phase) to the experts (⑤). After the transfer is complete, the next working phase starts.

The transfer of the new problem description can easily become a performance bottleneck as it is usually very large. Using standard transfer protocols such as TCP/IP can severely limit the performance of the system when the number of participating agents in a team exceeds a certain threshold because TCP/IP allows only for the sequential transfer of data to several recipients. Clearly, this is not acceptable and so means must be found to speed up the data transfer between the supervisor and the agents at the end of a team meeting. To overcome the performance bottleneck, the TEAMWORK LIBRARY implements a broadcast mechanics on the basis of IP datagrams. The purpose of the broadcast protocol is to provide a transparent, convenient and easy interface to the use of broadcasts. The approach for the use of broadcasts should be the same as for the stream sockets.

During protocol execution, one agent has an outstanding position called the *actual sender*. The actual sender is the only agent that is allowed to send broadcast messages at a certain time, but of course the sender can change over time.

The data transmission from the sender to the receivers is symmetric, ie. each send operation for a specific amount of data corresponds to a receive operation for exactly the same amount. Furthermore, the data transfer between two agents using broadcasts is done in a buffered manner, ie. a call to a send/receive operation may block the agent either because the data buffer is full and has to be flushed or it is empty and the new contents must be read.

During the protocol execution, the most important tasks for the agents are order preservation and loss detection of the packets sent. Broadcasts messages are unreliable datagrams and are therefore susceptible to duplication, order confusion or even loss. To illustrate how the TEAMWORK LIBRARY handles these problems, we will now discuss how the protocol works in detail.

Figure 6.13 illustrates how the transfer of a particular amount of data is organized in a *broadcast session* that is in turn subdivided in several *subsessions* that themselves consist of a *transmission* and a *acknowledgment/retransmission* phase.

The first thing to do for the current sender is to check whether all receivers are ready by waiting for a confirmation message from each receiver. After all re-

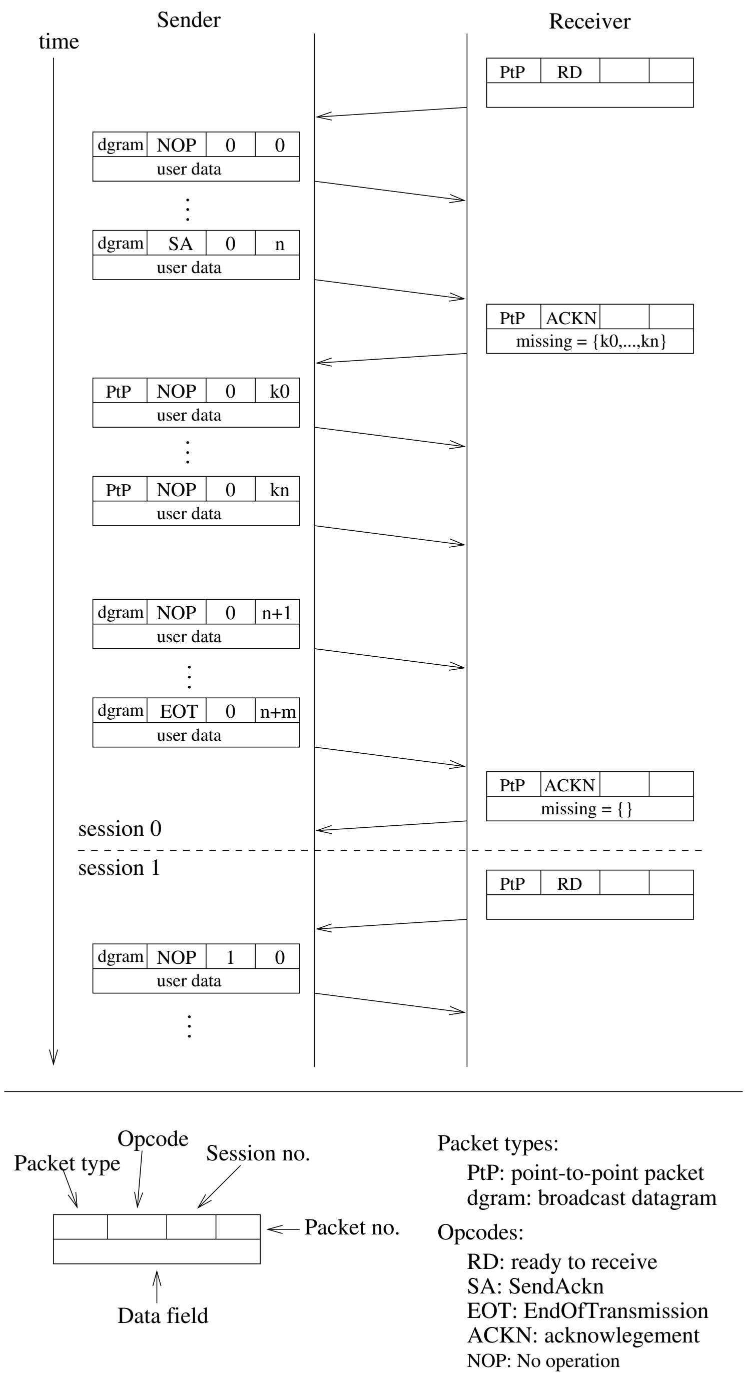


Figure 6.13: Broadcast Protocol

ceivers have confirmed to be ready, the data transfer can begin. The sender knows how many bytes have to be sent and from that value it can calculate the total number of packets needed to flush the entire buffer. This number is used to estimate the best number of packets to be sent before requesting an acknowledgment. The estimation will lie between a fixed lower bound and the total number of packets needed and it denotes the size of the next subsession. The estimation algorithm is based upon prior data about packet duplication or packet losses and thus implements an adaptive protocol.

After selecting the best size for the next subsession, the sender starts to transfer the proposed number of broadcast datagrams where each of these datagrams has a header and a data field. The header field contains the subsection and a packet number and an optional operation code to trigger certain actions on the receivers side. The two numbers in the header make it possible to identify every broadcast message during an entire Teamwork application uniquely. The receivers read the datagram messages and the header fields tell them what to do with an incoming packet.

The last packet of every subsession contains the header opcode `SendAckn` (or, if it is the last packet of the session, the opcode `EndOfTransmission` – which implies a `SendAckn`) that tells the receivers to enter the acknowledgment phase. This packet is protected against loss by the use of a timer, ie. when the sender realizes that acknowledgments are missing after a certain period of time, it will assume that at least one receiver did not get the packet containing the acknowledgment request. The last packet is then repeated until all receivers have sent their acknowledgment or until a maximum number of retransmissions was performed. In the latter case, the protocol issues an error message and abort because it assumes some irreparable damage at one of the receivers.

The acknowledgment scheme that is used in the TEAMWORK LIBRARY is called a *negative acknowledgment* scheme which means that the receivers keep track of the packets that were received and that the receivers inform the sender in their acknowledgment message which packets they did not receive. In a *positive acknowledgment* scheme, on the other hand, the receivers report which packets they got and the sender decides which packets must be retransmitted. After all the missing packets have been sent to the receivers, a new subsession begins or the transmission loop ends because the entire buffer was transferred.

6.2.5 Society View

The structure of the agent society within a typical Teamwork application is shown in Figure 6.14. The overall structure is hierarchical with the supervisor being the head that controls the subordinate agents. Each of the task specific

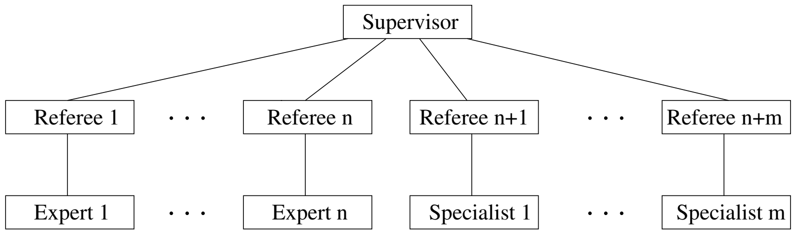


Figure 6.14: Team Structure

agents on the lowest level (experts or specialists) have an associated referee that evaluates their work in each team meeting and that is responsible for selecting good results to be forwarded to the supervisor.

In (KRONENBURG, 1995), an extension of the basic teamwork approach is presented that focuses on hierarchical team structures. A *team tree* can be built by allowing a team to have one or more *subteams* (which can have subteams as well). These subteams can be regarded as extended specialists as they can be used to solve subproblems of the main problem which are too hard for a single expert or specialist. Another field of application for subteams is a more administrative one. Subteams can be used to eliminate redundant information in order to reduce the memory usage of a teamwork application. Since the teamwork approach was designed for large search spaces, such a "garbage collection" is often needed to make a problem solution possible.

The subteam extension, however, is not yet implemented in the teamwork library, but some effort has been taken during the design phase of the library to make the later integration of the subteam extension as easy as possible.

6.2.6 Architectural View

6.2.6.1 System Architecture

In this section, we will characterize the required system architecture according to the scheme given in Section 5.7.1.

Entities The only entities that occur in the TEAMWORK LIBRARY are the agents that play the different roles according to the teamwork paradigm.

Control flow The control flow is fully determined by the Interaction view.

Information flow dto.

Agent management The agent management uses a proprietary mechanism that is executed by the initial team supervisor. Upon start, the initial agent becomes the supervisor and reads a team configuration file. Then, the supervisor launches an ID server that can be accessed by any subsequent supervisor in order to obtain unused unique identification numbers. Next, the supervisor starts new agents on the specified host computers for each teamwork agent. The newly created agents immediately connect to the supervisor and establish the communication network to the other agents.

Communication model The communication model of the TEAMWORK LIBRARY is based on proprietary TCP/IP messages that thus do not use any standard agent communication language.

Agent architecture The architecture of the Teamwork agents is discussed below.

Database design The TEAMWORK LIBRARY itself does not need any central databases, applications specific databases are not subject to this view.

External components/devices No external components or devices are used in the TEAMWORK LIBRARY.

6.2.6.2 Agent Architecture

Reasoning capabilities The reasoning capabilities of individual agents must remain unspecified because they are domain specific.

Resources limitations Each teamwork agent should be equipped with the computational resources of a Unix process in order to enable it solve demanding tasks in the problem domain.

Control flow The control flow within a teamwork agent is sequential because the role activations follow the ordering defined by the teem meeting protocol from Section 6.2.4.

Knowledge handling The knowledge representation of the agents depends on task-specific aspects that cannot be covered by a generic framework such as the TEAMWORK LIBRARY.

Autonomy The basic idea of the teamwork approach aims at distributed search in a large search space following a rather strict computational model that does not support agent autonomy in any way.

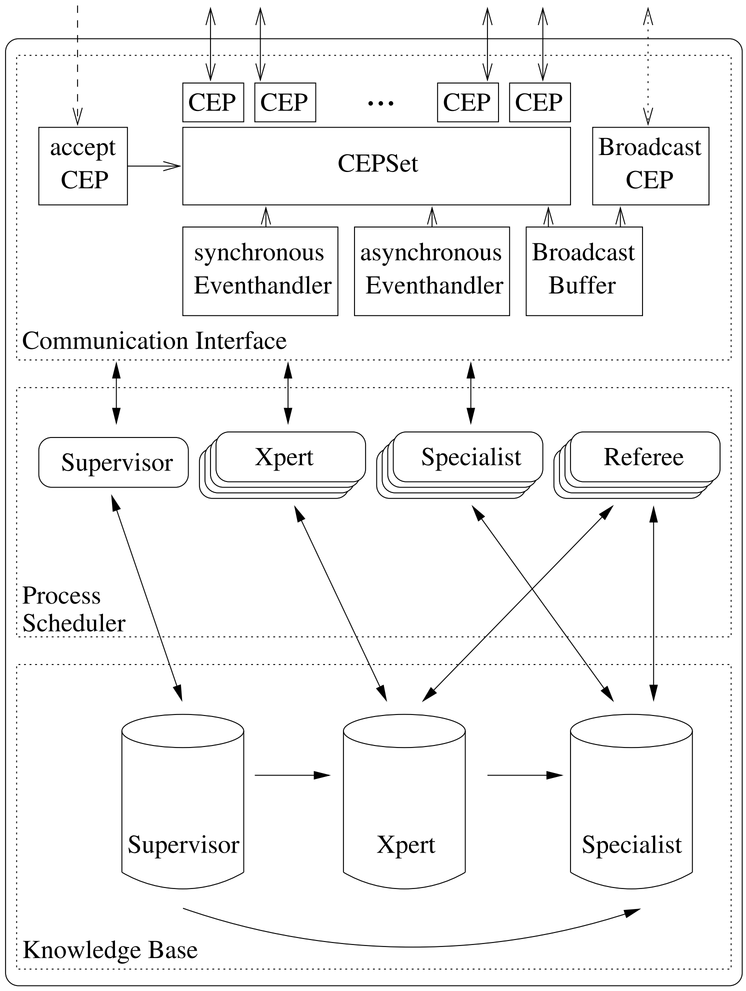


Figure 6.15: Teamwork Agents

User interaction No user interaction is intended for the agent since the teamwork approach is a batch-oriented search scheme.

Temporal context The lifetime of the teamwork agents will typically range only over a few hours and will thus not require any persistence mechanisms. However, if persistence is required in a particular application, the necessary mechanisms must be provided by the library user.

Rationality This aspect of the agent architecture is application specific and therefore not covered by the generic agent architecture.

The agent architecture of the TEAMWORK LIBRARY as shown in Figure 6.15 consists of three major parts:

- The *communication interface* of the agent architecture provides several mechanisms at different levels of abstraction to communicate with other agents of the application. AT the lowest level, normal TCP/IP stream connections are provided. The communication interface implements the standard connection establishing process by offering an accept communication end point (CEP) to which the other agents can connect. A new CEP is then created to handle the messages that are exchanged via the new connection. At a higher level of abstraction, these low level facilities are used to implement more complex forms of message exchange. One of these forms is the broadcast protocol based on IP datagrams that was presented earlier in this section, the two others are synchronous and asynchronous event mechanisms. A synchronous event mechanism assumes that the event subscriber expects an incoming event and dispatches this event according to a pre-defined handler. Asynchronous events differ from this by interrupting the subscribes control flow, dispatching the event and then resuming the control flow from the point where it was interrupted.

- The *process scheduler* that is implemented in the teamwork agents activates a particular role during the working phase according to the team structure that is provided by the current team supervisor. It is also responsible for switching between the roles of the working phase and those of the assessment phase and must thus know which expert or specialist is assessed by which referee.

- The *knowledge base*, finally, contains the domain knowledge that the agents use during the working phases and the specific knowledge of the supervisor that is needed during the team meeting. The referee roles can access the expert or the specialist part of the knowledge base in order to compute the short report and to select the good results for the long report.

6.2.7 System View

6.2.7.1 User Interface

Due to the generic nature of the TEAMWORK LIBRARY, the user interface that is provided together with the library must be limited to task-independent aspects of the final application. Thus, we have included a number of external tools that allow the developer and the user to monitor and analyze the system activities of a typical teamwork application. The tools that are provided with the TEAMWORK LIBRARY are as follows.

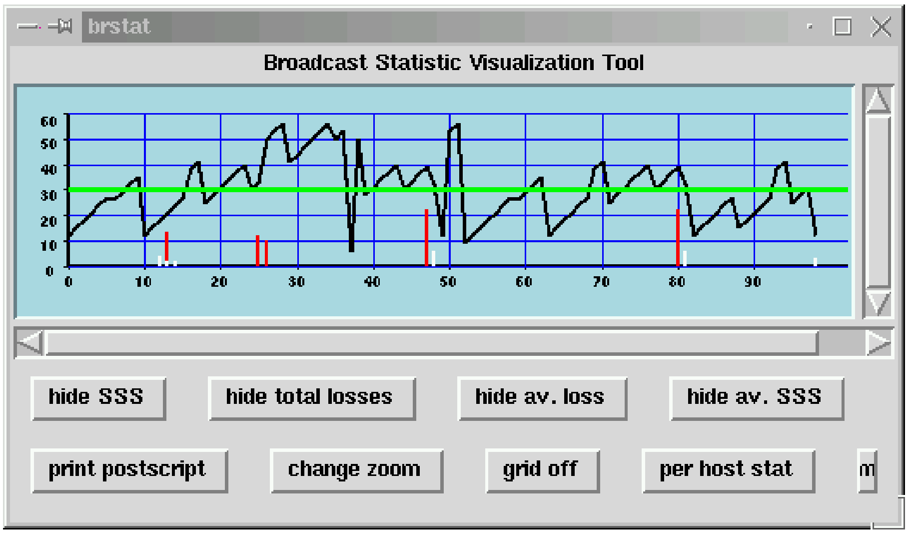


Figure 6.16: Broadcast Statistic Tool

The *Agent Monitor* shows the current state of each agent and the incoming and outgoing messages according to the Teamwork protocol described in Section 6.2.4. The second tool that is provided by the TEAMWORK LIBRARY is the *Society Monitor* that is used to visualize the current collection of agents that constitute a Teamwork application. The agent as well as the society monitor that were provided with the first version of TEAMWORK LIBRARY were strictly text based and have been extended in subsequent work (KÜNZEL, 1997). The extension also includes several tools to manipulate the behavior of individual agents or the agent society and a collection of generic base methods to support the development of a task-specific user interface.

A graphical tool that was provided with the original release of the Teamwork Library is the *Broadcast Analyzer* shown in Figure 6.16. This tools allows the system developer to perform a post-mortem analysis of the broadcast traffic during a teamwork application. The collected information can then be used to optimize the parameter setting for the broadcast protocol that was presented in Section 6.2.4.

6.2.7.2 Performance Engineering

Performance Engineering of the TEAMWORK LIBRARY was mainly focused on the communication subsystem of the agent architecture. To illustrate the performance engineering process, we will now present the estimation algorithm for the size of the next subsession during a broadcast session. This estimation algorithm uses statistical data that is collected in earlier subsessions to estimate the next value.

- The average packet loss during the last *averageLossHistory* subsessions. To obtain this value, a weighted arithmetic mean of the packet losses of the last *averageLossHistory* subsessions is computed. A weighted mean is used because the losses recorded in earlier subsessions should be taken less into account then more recent data. The weighted average mean is computed according to the following formula.

$$avLoss = \frac{\sum_{i=act-averageLossHistory}^{act-1} \left((\frac{1}{2})^{act-i-1} \cdot \sum_{j=0}^{numRecv} (loss_j(i)) \right)}{\sum_{k=0}^{averageLossHistory-1} (\frac{1}{2})^k} \tag{6.10}$$

  In equation 6.10, act is the number of the actual subsession, $numRecv$ holds the number of receivers and the function $loss_j(i)$ returns the number of packets lost by receiver j in subsession i. Please note, that the sum is not evaluated if i is less then 0 and note also that if the divisor of equation 6.10 is 0, $avLoss$ is assigned a value of *averageLossDefault*.

- The absolute number of lost packets containing opcode `SendAckn` or `EndOfTransmission` during the last *acknReqLossHistory* subsessions. This value is critical for broadcast performance because every lost acknowledgment request packet implies a delay (to detect that it was lost) and a retransmission. Therefore, special activities should be taken if this value exceeds a certain bound. The absolute number is computed to

$$absReqLoss = \sum_{i=act-acknReqLossHistory}^{act-1} \sum_{j=0}^{numRecv} acknReqLoss_j(i) \tag{6.11}$$

  Again, act holds the number of the actual subsection, $acknReqLoss_j(i)$ returns the number of packets lost by receiver j in subsession i. As in in equation 6.10, the sum is not evaluated if i is less then 0.

- The average size of the last *avSSSHistory* subsessions. Again, we use a weighted arithmetic mean to reduce the influence of older data.

$$avSSS = \frac{\sum_{i=act-avSSSHistory}^{act-1} \left(\left(\frac{1}{2}\right)^{act-i-1} \cdot sss(i) \right)}{\sum_{0}^{avSSSHistory-1} \left(\frac{1}{2}\right)^{k}} \tag{6.12}$$

In equation 6.12, $sss(i)$ returns the size of subsession i. If i does not denote a valid subsession number (e. g. if $act - avSSSHistory < 0$), the return value is ignored in the computation.

Now we will demonstrate how these values are used to determine the major parameter of the broadcast buffer during the sending phase. The size of the next subsession is chosen according to

$$nextSSS = \begin{cases} lastSSS + increaseUnitSize & avLoss \leq littleLoss \\ avSSS & littleLoss < avLoss \\ & \quad \leq mediumLoss \\ avSSS - avLoss & mediumLoss < avLoss \\ & \quad \leq highLoss \\ avSSS - avLoss - decreaseUnitSize & avLoss \geq highLoss \end{cases} \tag{6.13}$$

Another parameter that determines the senders behavior is the *send delay.* If a send delay is activated, the last packet of every subsession which contains contains the acknowledgment request is sent twice, with a delay of a few microseconds between the two send operations. This helps to reduce the number of acknowledge request losses because the delay enables the receivers to process some datagrams received earlier thus freeing internal buffer space for more incoming packets. The delay option is turned on if the number of acknowledge request packets lost exceeds the value of *absAcknReqLossBound.*

This was only one example of the Performance Engineering activities that have been applied to the TEAMWORK LIBRARY. Further – and more detailed – information about performance aspects of individual parts of the library can be found in (LIND, 1996B).

6.2.7.3 Deployment

Deployment planning was a major concern of the TEAMWORK LIBRARY is a was intended to be usable on a broad variety of hardware platforms that support the basic requirements that were outlined in Section 6.2.2. Therefore, the final release of the Library contained a generic Makefile that was configured by the library user to fit the particular needs of an installation site. The details of the deployment process are beyond the scope of this overview, the interested reader may refer to the reference manual that is shipped with the library (LIND, 1996B).

6.3 Summary

We have presented two additional case studies to demonstrated the usefulness of the MASSIVE method in real world applications. The multiagent solution for the intermodal route planning service within the PTA domain has shown how industrial standards such as FIPA can be easily integrated into the design of a multiagent system. The TEAMWORK LIBRARY is a collection of software components that support the development of multiagent applications according to the teamwork distributed search paradigm. The MASSIVE method turned out to work for such a general setting that lacks a specific task as well as for more specific domains.

Chapter 7

Related Work

THE RELATED RESEARCH WORK that is discussed in this chapter comes from two fields: development methods for multiagent systems that have been proposed either by industrial projects or by scientific research and frameworks or tool-kits for agent-based applications.

7.1 Development Methods for Multiagent System

Some of the approaches for the development of multiagent systems were specifically designed as general-purpose methods while others are the result of the analysis of the development process of a particular system. We will present each of them according to a fixed scheme that makes it easier to assess individual aspects.

The scheme that we will apply to each method presented in the course of the next sections is as follows:

Scope This aspect describes the overall goal of the method and the parts of the life cycle that are covered by the method. Furthermore, we will also indicate the origin of the method an whether a case-study was supplied in the original material that presented the idea.

Models and Representation After the first more introductory part, we will then become more technical and present the main models of the method as well as their purposes. We will also briefly discuss the different levels of abstraction that are supported and whether and how the static and dynamic aspects of the systems are modeled.

Process Model Here, we will outline the process model that is applied in order to construct the models discussed in the previous aspect and explain the main steps in the proposed life cycle model.

Assessment This part of the description will summarize the major advantages and the shortcomings of each method. We will also apply a ranking scheme for general properties as they appear to the author. The properties that are evaluated are

Generality ie. the range of multiagent systems that is supported or the commitment to a particular technology or agent architecture,

Flexibility which covers aspects such as the extensibility of the method or support for different process models or tools,

Granularity captures the level of detail by which system aspects can be modeled and the supported levels of abstraction,

Formality eg. the use of formal methods or well defined semantics of modeling elements and finally

Tool support which summarizes available tools for a particular method.

There is a wide variety of methods and the selection of ideas presented in the subsequent sections can only show a small amount of the full picture. However, we have tried to find those methods that either have a unique view onto the target system or that are reasonably general to be applied in a general context.

7.1.1 Burmeister

Scope The approach presented in (BURMEISTER, 1996) is a development method for multiagent systems based on object-oriented techniques. The method is limited to systems of cooperative, directly communicating software agents. A short case study is provide to illustrate the basic ideas but no reference for an application of the method in an industrial application is given.

The approach covers the analysis and design phase of the software development process and it does not assume a particular life cycle model, although the process model presented later suggest that a linear life cycle model is used by the authors. The distinction between the analysis and design phase are somewhat fuzzy and seem to make it difficult to organize them in a sequential process.

Models and Representation The Burmeister approach uses three models to describe the entities and their interconnections within a multiagent system. The *Agent model* describes the internal states, ie. the goals or plans, of the agents and thus models the *dynamics in the small.* The *Organizational model*, on the other hand, defines the relationships among agents and agent types. These relationships include the inheritance hierarchy, the roles that occur within the system and the relationships among agents based on the roles they play. The objectives of this model can be two-fold in that it is either used to structure the system into several sub- systems or to represent a real organization. The *Cooperation model*, finally, describes the *dynamics in the large* of the target system, ie. the interactions among agents. The interactions (limited to communication) are modeled by protocols based on KQML messages. Furthermore, the method implicitly assumes a cooperative setting (ie. benevolent agents).

The method does not suggest any special from of representation, however, CRC (classes-responsibilities-collaborations) cards are recommended for establishing the the agent model and interaction or collaboration diagrams known from OOD are suggested for the cooperation model

Process Model The method defines a three step process model as show in process Model 10 that corresponds to the three models mentioned above. Each of these steps should be executable in parallel as the models are claimed to be independent. It seems however, that the author assumes these steps to be completed in the given order.

Assessment The method is conceptually clear and intuitive and provides an easy access to the basic concepts because of the relation to OO techniques. However, it is not very detailed and some of the claimsare not substantial. Most striking is the claim that the different models are independent and can thus be developed separately but for example, the cooperation model can only be developed after the entities (agents) that interact are defined and thus the cooperation model depends on defining the agents first.

Additionally, the activities within a single phase of the process model are somehow incoherent. For example, in the organizational model, the role assignment to particular agent types is made before the agent types are defined. Furthermore, the method pays only little attention to the interoperation of the system with its environment.

Process Model 10 Burmeister

(1) **Agent model**

(1.1) Identify and characterize agents and environment
This step mainly aims at the definition of the entities that should become the agents of the multiagent system and suggests to identify the active entities in the problem description.

(1.2) Define motivations of the agents
The motivations of an agent are its interests, preferences, responsibilities, long-term goals etc.

(1.3) Define behaviors (predefined plans)
The method defines the behaviors of an agent in terms of plan schemes that describe how a particular goal can be reached.

(1.4) Define knowledge and beliefs
In this step, the knowldege and beliefs that are used by the agent to fulfil its tasks are defined.

(2) **Organizational model**

(2.1) Identify roles within the scenario
In this step, not only the roles that occur within the system are identified, these roles are also mapped to the agents or agent types that are supposed to play the roles.

(2.2) Build inheritance hierarchy
Agents are classified according to their knowledge, beliefs, motivations and behaviors in order to set up an inheritance scheme similar to OO design that describes the static interconnections between agents.

(2.3) Structure roles within the organization
In this step, the inter-role dependencies are modeled on the basis of the intended structure of the multiagent system or on the basis of a real organization that is to be modeled by an artificial system.

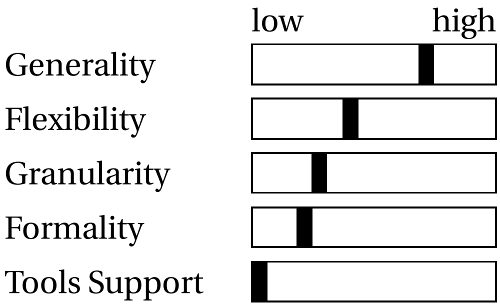
(3) **Cooperation model**

(3.1) Identify cooperation partners
For each joint task, the agents that are necessary to complete it are identified and the type of cooperation (eg. Resource sharing, synchronization etc.) is defined.

(3.2) Identify message types
The method suggest to use KQML messages to describe the messages that are exchanged between the agents.

(3.3) Define cooperation protocols
The patterns of interaction between the agents are described by specifying the cooperation protocols that are needed to complete joint tasks of different agents.

This is however, an important issue if the target system is to be used in a organizational context and should be explicitly dealt with. The overall value of the method is hard to decide as it is not clear if the method has been tested in an industrial project.



7.1.2 Kinny and Georgeff

Scope (KINNY AND GEORGEFF, 1996) present an object-oriented development method for BDI (RAO AND GEORGEFF, 1995) agents and multiagent systems of BDI agents. The approach covers the analysis and design phases of the software development process and assumes no particular life cycle model.

Models and Representation The proposed method is centered around basic models that are in turn subdivided into several sub-models. The first basic model is the *External viewpoint* that describes the more coarse-grained aspects of the system and its dynamics in-the-large and consists of two sub-models. The *Agent model* defines the inheritance hierarchy among agent types and the instance model, eg. role multiplicity, of the agent society. The *Interaction model* describes the responsibilities of each agent class, the services it provides and the message model in terms of syntax and semantics of the exchanged messages.

The *Internal viewpoint*, on the other hand, is concerned with the design of individual agents according to the BDI paradigm. The internal viewpoint consists of three sub- models: the *Belief* and the *Goal model* describe the domain-specific belief and goal set, respectively and the *Plan model* defines the actions that must be taken by the agents in order to achieve a particular goal.

The models that are developed in the method are represented as enhanced UML class diagrams for belief class models and finite state automata for plan models. Furthermore, formal definitions for beliefs and goals are provided but the system dynamics are not explicitly modeled in the method.

Process Model The proposed process model is a sequential model that consists of two major steps as shown in Process Model 11.

Assessment The proposed method provides a set of models that can be used

Process Model 11 Kinny and Georgeff

(1) **Develop external models**

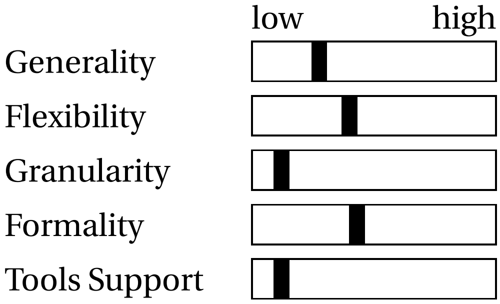
- (1.1) Identify roles
- (1.2) Define role responsibilities (= services)
- (1.3) For each service: identify necessary interactions, speech acts etc.
- (1.4) Define the agent hierarchy according to the previous steps

(2) **Develop internal models**

- (2.1) Analyze means of achieving goals
- (2.2) Build the beliefs of the agents

to describe most aspects of a multiagent system. The object-oriented approach supports developers that are familiar with the basic concepts and also leads to a modular design.

However, the method is limited to BDI agents and provides only a very sketchy process model that leaves a lot of questions unanswered, eg. how to identify the roles, etc. This is the crucial factor in the entire design process as all of the models that are developed in subsequent steps depend on these decisions. If, however, no guidance is given on how to actually find the fundamental roles, the entire design may fail because of wrong decisions at a very early stage of the development process. Furthermore, the proposed process model is very high level and does not provide guidance on how to perform the different steps.



7.1.3 DESIRE

Scope DESIRE (Brazier et al., 1996), (Brazier et al., 1997), (Brazier et al., 1998) is a conceptual framework for the analysis and design of multiagent systems, the extensive case study that is provided describes a project that is concerned with the management of an electricity transportation network. Management means in this respect the monitoring, fault diagnosis and maintenance planning of the network. The full system consists of seven agents out of which four are presented in detail in the case study.

The method covers the analysis and the design phase of the software development life cycle and assumes no particular process model although a linear, sequential model is used in the case study.

Models and Representation The method takes a compositional view on agents and systems: the entire functionality of the system is designed as series of interacting, task-based, hierarchically structured components. Each task can either be primitive or it can be composite; a task hierarchy is constructed by applying a recursive top-down decomposition process on the initial system task. The compositional view on agents sees an individual agent as a collection of task-solving units (components). The terms "task" and "component" seem to be used interchangeably in the DESIRE documentation.

The DESIRE framework proposes two models that should be specified by the system designer. The *intra-agent* model contains the expertise descriptions for domain tasks, the knowledge requirements and the reasoning capabilities for solving these tasks. The *inter-agent model*, on the other hand, describes the expertise to perform and guide coordination, cooperation and social interaction among agents.

DESIRE is centered around the generic agent architecture depicted in Figure 7.1 that consist of five elements. The *Task Control Unit* encapsulates the control knowledge of the agent, ie. the relations between the components and the activation schemes that are enacted during the agents operation. The *Process Control Unit* is responsible for coordinating the agents reasoning processes, the *Communication Control Unit* manages for the social aspects of agent by handling incoming and outgoing messages, the *World State Monitor* constantly updates the world model of the agent by incorporating perceptions into the current model and a task-specific *Task Execution Unit*, finally, implements the particular skills of different agents.

The system dynamics are modeled by information links between the problem solving units, they can thus be used to model intra-agent information flow as well as inter-agent information flow. The concepts used in DESIRE can be formalized using temporal logic.

Process Model DESIRE proposes a five step process model as shown in Process Model 12 is executed in sequential order.

Assessment The major advantage of the DESIRE method is that is uses a unified view on all entities within the system. The holonic approach makes

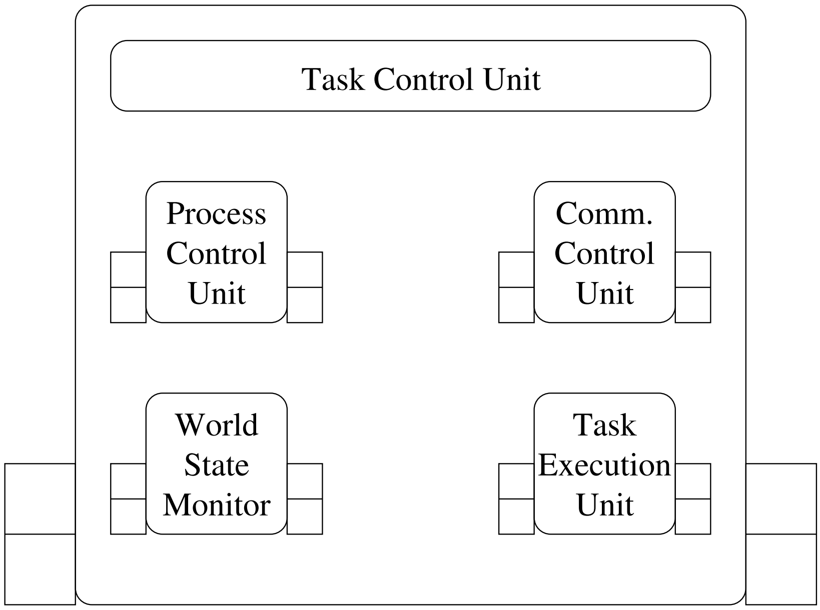
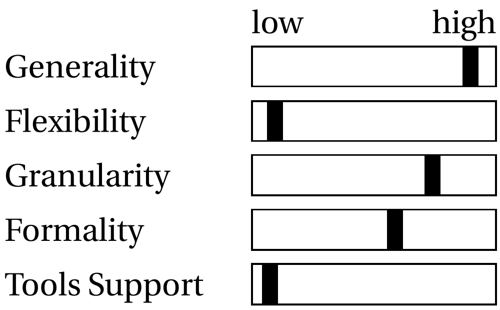


Figure 7.1: The Generic DESIRE Agent Architecture

it easier for the designer to think in functional terms rather then in agent-oriented concepts. However, the commitment of the method to a specific agent architecture and the strict process model limit its general applicability to a broader range of problems.



7.1.4 MAS-CommonKADS

Scope MAS-CommonKADS (IGLESIAS ET AL., 1997) is a general purpose multiagent analysis and design method based on the CommonKADS design method for knowledge-based systems (KBS). CommonKADS supports most aspects of a KBS development project, including project management, organizational analysis, knowledge acquisition, conceptual modeling, user interaction, system integration, and design. The method is result

Process Model 12 DESIRE

(1) **Task decomposition**

- (1.1) Define overall task hierarchy.
- (1.2) Define individual task descriptions (inputs, outputs, meta-object relations, eg. which task reasons about which other task, etc.).

(2) **Information exchange model**

- (2.1) Specify the information links between components.
- (2.2) Define input/output relations, ie. the input of which component is the output of which other, a translation mechanism enables to designer to use different languages for the task internal knowledge representation.

(3) **Sequencing of sub-tasks**

- (3.1) Describe task control knowledge (temporal order of tasks, effort to solve the task, etc.).
- (3.2) Specify intra- and inter-agent sequencing of tasks

(4) **Sub-task-delegation**

- (4.1) Assign tasks to agents.
- (4.2) Perform functional grouping of tasks.

(5) **Knowledge structures**

- (5.1) Define structures for the domain knowledge by providing concepts that describe the distinguishable objects in the domain and relations that describe how the concepts relate to each other.

---

oriented rather than process oriented and it describes KBS development from two perspectives. The *result perspective* captures a set of models that describe different aspects of the KBS and its environment. These models are continuously improved during a project life-cycle. The *project management perspective*, on the other hand, features a risk-driven generic spiral life-cycle model that can be configured and adapted to the particular project.

CommonKADS extends this framework by adding techniques from object-orientation (OMT, OOSE, RDD) and protocol engineering (SDL, MSC) and tailoring them towards the specific requirements of multiagent systems. The design method is primarily suited for systems of directly communicating software agent. An example application called the "Travel Agency" is provided in (IGLESIAS ET AL., 1997). In the problem domain, a user requests a flight from the system and system finds the cheapest available flights with the lowest possibility of delays by querying the available flight carriers. The method covers the analysis and design phase of the software development process and proposes a risk-driven, component-based ap-

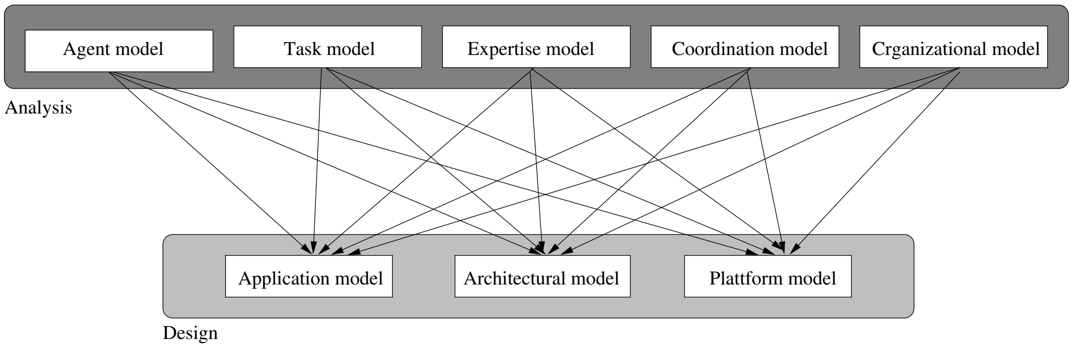


Figure 7.2: Hierarchical Set of Models

proach but no details of how this process can be implemented are provided.

Models and Representation The MAS-CommonKADS method is centered on a set of different models as shown in Figure 7.2 that are used to describe the target system. During the *Analysis phase* of the development process, the first sub-set of models to be constructed contains the *Agent model* that describes the agent characteristics such as the reasoning capabilities, the skills of the individual agent or the agent architecture to be used. Secondly, the *Task model* defines the task decomposition within the problem domain and contains the task descriptions. Thus, the task model is closely related to the *Expertise model* where the basic knowledge requirements for carrying out the tasks are described. The *Coordination model* captures the conversations between agents, their interactions and the protocols used. The context of this coordination process is described by the *Organizational model* in terms of the structure of the agent society or the agent hierarchy. The *Communication model*, finally, covers the user interface of the target system.

During the *Design phase* of the development process, these initial models are used as the basis for the design model, which is composed out of the *Application*, the *Architectural* and the *Platform model*. The Application model captures the overall structure of the target system, the Architectural model defines the networking and telematic requirements and the Platform model, finally, describes the development platform for the system.

Process Model 13 MAS-CommonKADS

(1) **Conceptualization**
This phase yields a preliminary description of the problem area and an informal requirements definition.

(2) **Analysis**
This step performs a requirements specification by developing several of the above mentioned models.

(2.1) *Agent model* In this step, initial instances of the agent model are created by performing a semantic analysis of the problem description and applying conceptual distance heuristics based on the determination of shared knowledge structures, organizational or geographical distribution etc. to the initial problem description.

(2.2) *Task model* The goal of this step is the task decomposition and goal determination of the agents using a top-down decomposition. The resulting task descriptions include the name, input/output data, control structures, frequency, preconditions, required skills of the performer etc.

(2.3) *Coordination model* This model describes the interactions and coordination protocols by specifying prototypical situations identifying messages and exchanged data in the form of SDL diagrams.

(2.4) *Knowledge model* This model contains the domain knowledge (declarative knowledge of the problem domain), the inference knowledge (what sorts of inferences are needed eg. diagnosis, assessment, search, etc.), the task knowledge (order of inference structures) and the problem solving method(s) (define the implementation of each inference type needed) that occur within the target system.

(2.5) *Organization model* The result of this step is a static description of the relationships between agents as well as the organizational structure of the enterprise before and after the introduction of the MAS, ie. the given structure of the enterprise and the target structure.

(3) **Design**
The design model refines the models of the analysis step and constructs the three design models mentioned earlier.

(3.1) *Agent network design model* This model describe the infrastructure (network, coordination facilities etc.) of the MAS, the network facilities (naming service, security, encryption, etc.), knowledge facilities (ontology servers, knowledge representation translators, etc.) and the coordination facilities (protocol servers, group management facilities, resource allocation, etc.) within the target system.

(3.2) *Agent design model* In this step, the most suitable architecture for each agent is determined. The method suggest a generic agent architecture that consists of a user-communication unit for the interaction between the agent and the user, an agent communication unit for the interactions among agents, a deliberation and reaction unit for the the reasoning of the agent and an external skills and services unit that provides the agents services to the other agents.

(3.3) *Platform design model* In this model the software and the hardware platform for each agent are selected and described.

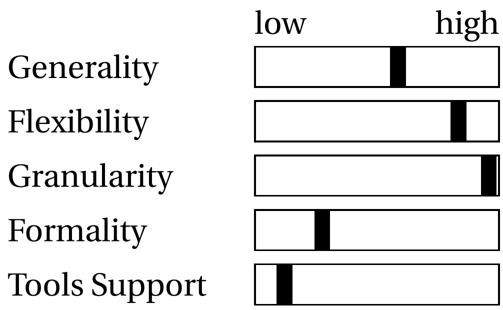
The method recommends a number of techniques to describe these models, for example

- *Use Cases* during the conceptualization phase to describe prototypical uses of the target system,
- *Class-Responsibility-Collaboration (CRC) cards* for the agent modeling process,
- *Event Flow Diagrams* and *Specification and Description Language (SDL) diagrams* for the coordination modeling and
- the *Conceptual Modeling Language (CML)* and *Inference Structure Diagrams* for knowledge modeling.

Process Model The proposed process model is shown in Process Model 13.

Assessment The MAS-CommonKADS method is very detailed and provides a conceptual framework for the analysis and design of multiagent systems that is based on an existing, working approach that is currently in use for the development of knowledge- based system. The method is not focused on the functional aspects of the target system, it also models the target environment and important aspects such as agent management aspects.

On the other hand, the method lacks a bit of an overall structure and sometimes appears as a collection of models with not explicit links between them. This makes it hard for the less experienced user to fully understand the model structure and to find the most suitable model for a particular aspect of the target system. Finally, the design method focuses on virtual systems with explicit message exchange and is thus of limited use for other types of applications .



7.1.5 Wooldridge, Jennings and Kinny

Scope (WOOLDRIDGE ET AL., 1999) propose a method for the development of multiagent applications with the following characteristics:

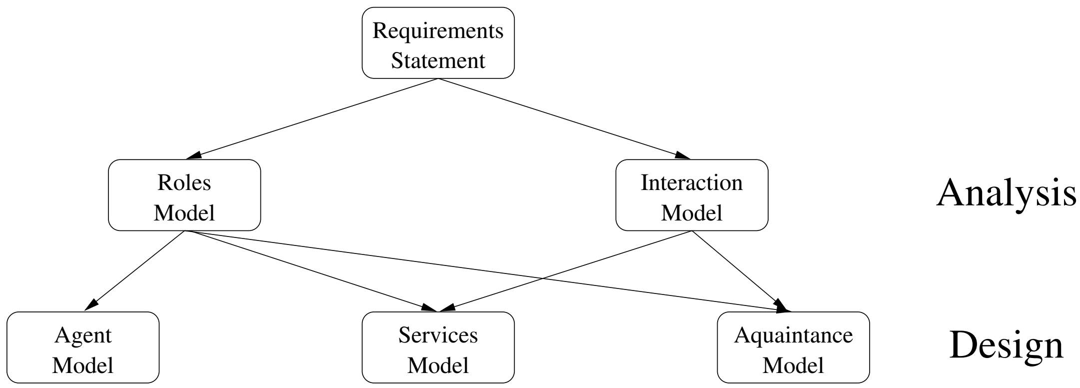


Figure 7.3: Model Relations (WOOLDRIDGE ET AL., 1999)

- The application is coarse-grained, ie. the computational resources that are available to a single agent should be comparable to those of a UNIX process.
- The problem domain is an optimization problem where all agents share the common goal to minimize or maximize some given measure. The problem domain does not contain true conflict situations that must be solved.
- The agent society is heterogeneous, ie. agents may be implemented in different programming languages or on different platforms.
- The multiagent system is relatively small, ie. less then 100 agents.
- The agents do not exhibit mobility.

The proposed method covers the analysis and design phase of the software development process on a very high level. The resulting design is intended to be used as the input for traditional software engineering methods that refine the high-level design into a particular implementation.

Models and Representation The method features an integrated view onto the system as a computational organization of interacting roles. Roles are dynamically assigned to agents and an agent can play several roles at the same time. The main models that are developed in this method are discussed below, the relationships between the individual models are shown in Figure 7.3, the process model itself is outlined in Process Model 14.

The *Roles model* contains several abstract descriptions of an entities expected function. Each of these descriptions defines the *permissions* of the role, ie. the resources that *can* be used in the course of problem solving as well as the resources that *cannot* be used during this process. Resources in this respect are usually information or knowledge structures that are either available or not. Secondly, the role is defined by its *responsibilities*, ie. what it has to do within a well-defined context. The responsibilities are defined in terms of *liveliness conditions* that express what has to be done while the role is active and in terms of *safety conditions* that describe invariants that must be maintained at all time during the lifetime of a role.

The *Interaction model* describes the links between individual roles and the protocols that are used to interact. Each of these protocol descriptions is given by its *purpose*, eg. "information request" or "assign task", its *initiator*, ie. the role that starts the protocol, the *responders* that follow the protocol initiation, the *input* information that is used by the initiator, the *output* information that is supplied to the responders and the *processing* that is performed by the initiator.

The *Agent model* defines the *agent types* by aggregating roles, eg. to increase the efficiency of the application. The agent model does not support inheritance relationships between roles as they are regarded not necessary by the authors because the granularity of the application will seldom require other mappings then one-to-one mappings from role to agent type.

The *Service model* describes the functionality of an agent. Functionality in this respect refers to a single, coherent block of activities and is defined by the *inputs*, the *outputs* and the *pre*- and *post-conditions*. Note that this description is still on a very high level and does not prescribe any particular implementation of the functionality.

The *Acquaintance model*, finally, defines the *communication links* between agents, the goal of this model is to identify potential communication bottlenecks.

The method proposes some formal language to represent the liveliness and safety conditions of a role and some templates for role schemata and protocol definitions.

Process Model The process model that is used within the method consists of an iterative analysis phase that is followed by a sequential design phase. The relation of the models to these two phases of the development process are shown in Figure 7.3

As already stated above, the resulting design is thought as a high level description that must be refined in subsequent steps in order to yield to the

Process Model 14 Wooldridge, Jennings and Kinny

(1) **Analysis**
The goal of this phase is to understand the structure of the target system without referring to a particular architecture.

(1.1) Identify role prototypes

(1.2) Identify the protocols of each role

(1.3) Elaborate the roles model

(1.4) Go to 1a until finished

(2) **Design**
The goal of this phase is to transform the models from the first phase into sufficiently low level to serve as an input for traditional design processes.

(2.1) Create the agent model by aggregating roles

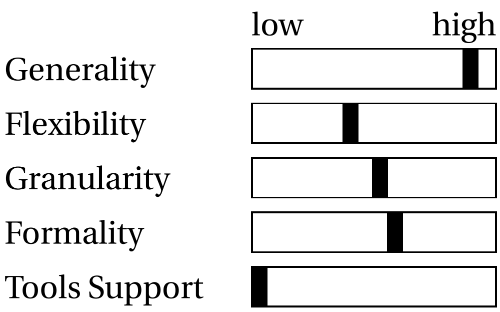
(2.2) Develop the service model

(2.3) Develop the acquaintance model

final design. The proposed process model is shown in Process Model 14.

Assessment The advantage of this method is that it provides a detailed set of models that capture almost all relevant aspects of the target system. It is also straightforward to apply even by unexperienced users. Furthermore, the method is not committed to any particular technology such as BDI agents etc.

However, the major problem is the attempt to define the method on top of other software engineering techniques. The potential problem of this idea becomes apparent if is not possible to find an adequate implementation for the high level design. The entire design (perhaps even the analysis) has to be revised in order to develop a model that can actually be implemented. This problem could be weakened if the proposed set of models contained a model of the environment of the target system but still this would bear potential problems if the refinement from a high level to a low level design and finally to an implementation is not straightforward.



7.2 Toolkits for Agent-Based Applications

In this section, we will review three tool-kits for the development of multiagent applications. These packages usually offer predefined structures or even tools that can be used by the application developer to build his or her own system using the provided structures that are common to most multiagent applications. All tool-kits discussed in this section are freely available and result from research work in the area, commercial frameworks have not been considered for this section.

7.2.1 SIF

The **S**ocial **I**nteraction **F**ramework (SIF) (LIND, 1998), (FUNK ET AL., 1998), (SCHILLO ET AL., 1999) is an open, easy-to-use testbed for multiagent applications. It is intended as a simulation tool for interaction-intensive societies and is – as opposed to most other testbeds – not limited to communication but supports various forms of interaction.

SIF provides an open simulation environment that allows the integration of agents with different architectures in distributed environments. The fine-grained action and perception modeling allow for the design of complex virtual worlds without any predefined semantics on percepts ("The world does not come labeled" (EDELMAN, 1987)). Furthermore, SIF supports rapid prototyping by providing off-the-shelf components for the simulation of a wide range of interaction environments and also provides flexible human user interaction through the use of avatars. SIF can be used on a number of different platforms because of using standard technology whenever possible eg. Java (SUN MICROSYSTEMS, 1999), VRML97 (THE VRML CONSORTIUM, 1997) or CORBA (OBJECT MANAGEMENT GROUP, 1999).

The interaction model of SIF is based on the **E**ffector-**M**edium-**S**ensor (EMS) Model (LIND, 1998) which is a conceptualization of the broad agent definition given in (RUSSELL AND NORVIG, 1995) that "An *agent* is anything that can be viewed as *perceiving* its environment through *sensors* and *acting* upon that environment through *effectors*." An example for applying this definition to human speech interaction is shown in Figure 7.4.

The speech apparatus of the agent on the left hand side of Figure 7.4 is the effector that triggers the medium which in turn delivers some perception to the sensoric unit of the agent on the right. Thus, the EMS paradigm captures a very natural model of interaction.

A more technical view onto the EMS model that describes the basic control cycle of an agent is shown in Figure 7.5. Whenever several agents are connected

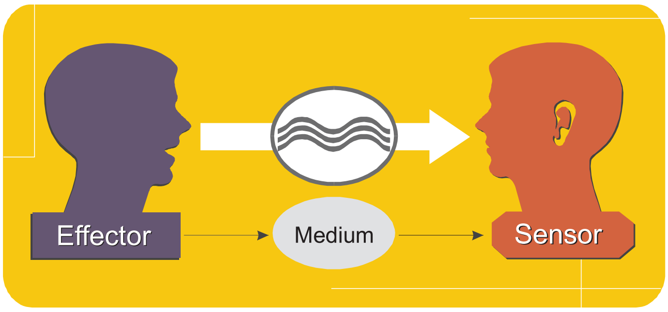


Figure 7.4: The EMS Idea

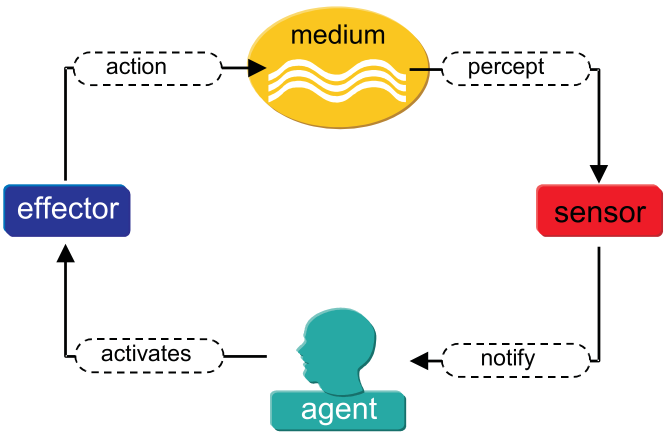


Figure 7.5: The EMS Model

to the same medium they can interact with each other by changing the state of the medium. The data-flow within the system is asynchronous due to the independence of the control flow of the agents from the control flow of the medium and the control flow is multi-threaded. Furthermore, the SIF framework implements a subject-oriented point-of-view because the agents locally perceive the world state.

The agents themselves need not have a specific architecture as long as the architecture fulfills a minimal conceptual framework that is shown in Figure 7.6. According to the minimal requirements, each agent must have its local thread of execution, the agent must implement a decision function that computes its next action on the basis of the agents current world model and action invocation

Figure 7.6: Conceptual Agent Model of SIF

must be performed asynchronously in order to prevent an agent from blocking the medium and thus the actions of other agents. Note that the action execution is not guaranteed to succeed, ie. an agent can issue a particular action activation to the medium, however, if the action is not valid in the context that is defined by the medium it will simply be ignored. It is therefore important for the agents to monitor their action execution and not to simply assume that an action will be executed once it is submitted to the medium.

The task of the medium is to model the environment from a static (entities) and dynamic (physics) point-of-view. Thus by designing the medium, the designer defines which entities can be found within a particular world and how the world changes according to action invocations of the agents. The basic control cycle of the medium is shown in Figure 7.7.

In each cycle, the medium receives the effector activations of the agents and places them in a queue of pending activations. Then it fetches the first of the pending activation requests and updates the world model according to the effects of the activation. If a particular action is not allowed because of the current state of the world, it is discarded. If, on the other hand, the world state has changed because of a valid action, the medium generates perceptions as the projections of the current world state onto the sensors of each agent.

The visualization of the world state is always modeled from the point-of-view of an agent. However, to enable the user to have a global model of current state, a special "omnipresent" agent that can perceive the full world state is provided. Uniformly modeling the the visualization devices as agents supports the use of difference technologies because it provides a well-defined interface between the medium and the visualizing agent. In the current implementation of an example application for SIF, a 2D Java-based visualizing agent and a 3D VRML-based visualizing agent were used to present to same world at the same

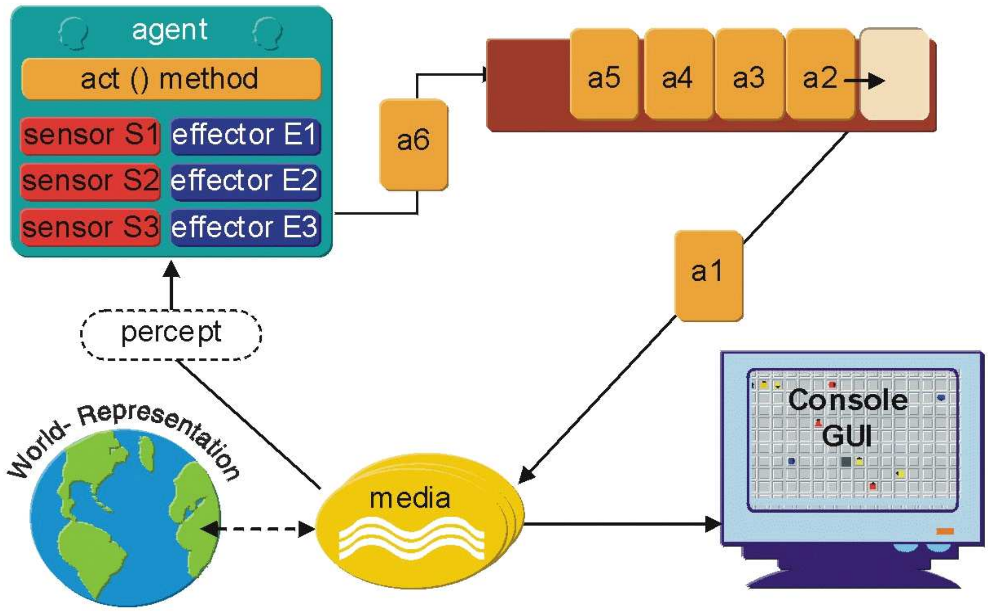


Figure 7.7: Information and Control Flow in SIF

time to different users on different machines.

In SIF, user interaction is achieved by using so-called *control pads* to steer the agent directly. The integration of user commands is achieved via an extra indirection cycle as shown in Figure 7.8: the agent has special effectors to perceive user commands that change its behavior accordingly. The additional indirection de-couples the agent and the user in a way that enables the agent to carry on with its normal operation when no user commands are given and to react to the user inputs whenever this is requested. This scheme provides the necessary flexibility to use the agent as an *avatar* that is autonomous to a certain degree and reliefs the user from having to specify every single behavior activation.

The SIF tool-kit has shown that it is a truly open testbed that can be used to implement applications with SIFAgents, InteRRaP agents or MECCA (STEINER, 1992) agents. The flexible visualization mechanism can be used to generate different views on the same virtual world and the control mechanism that is provided is well-suited for applications with semi-autonomous agents. SIF is platform independent and provides a library of pre-defined off-the-shelf components and algorithms for rapid prototyping and the underlying EMS paradigm allows for the definition of a large variety of different virtual environments. However, developing world models is a work-intensive task because of the fine granularity of the simulations.

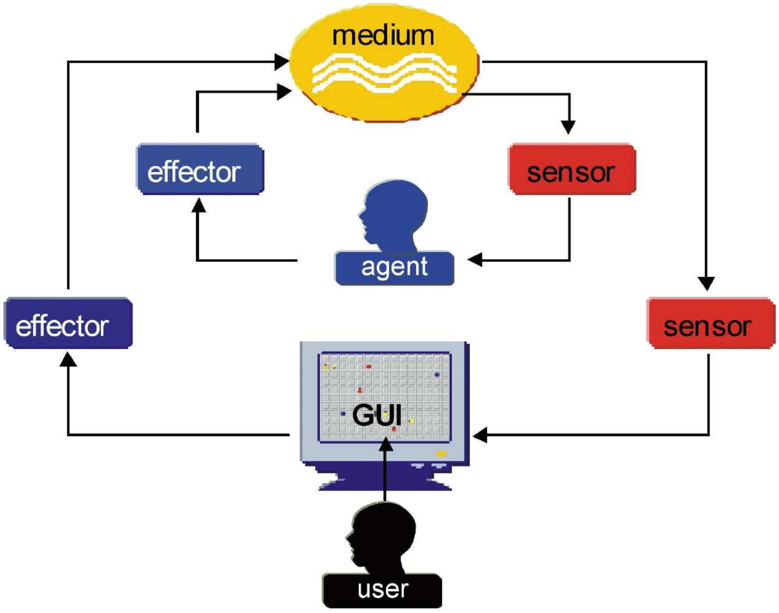


Figure 7.8: SIF User Interaction

7.2.2 ZEUS

The goal of the ZEUS system (NWANA ET AL., 1999) is to provide a design method and tool support for the engineering of distributed multiagent applications. The provided tools all encompass the direct-manipulation metaphor and allow the designer to use drag-and-drop technology to assemble the application from pre-defined components.

The tool-kit allows the designer to specify models for different types of agents, for the organizational structure of agent societies and for negotiation models. The negotiation models are either pre-defined or the can be build by the designer if no appropriate pre-defined model is available for a particular task.

ZEUS agents all have the same architecture as shown in Figure 7.9 that consists of the *definition layer* that implements the reasoning and learning capabilities of the agents, the *organization layer* that manages and maintains the relationships with other agents and the *coordination layer* that is responsible for the inter-agent coordination and that also contains the negotiation knowledge.

Two additional layers are provided that are responsible for the technical framework of the agents. These additional layers are the the *communication layer* that provides the communication facilities for the communication with

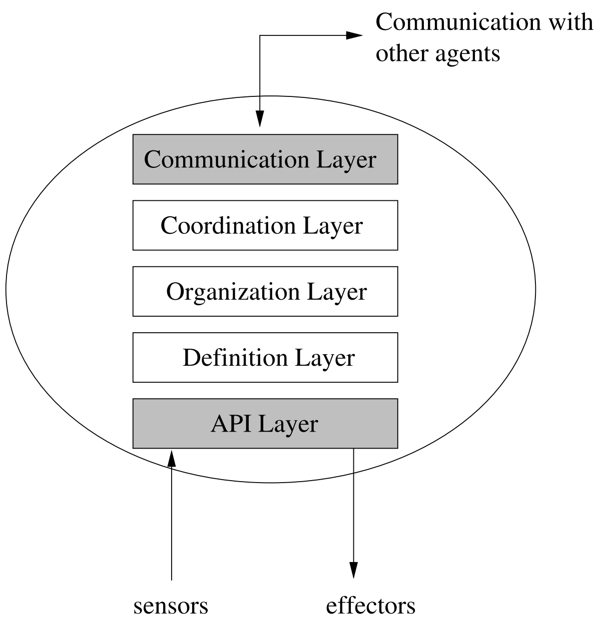


Figure 7.9: ZEUS Agent Architecture (NWANA ET AL., 1999)

other agents and the *API layer* that serves as the world interface of the agents.

The ZEUS tool-kit is mainly a collection of editors that can be used to develop multiagent applications according to the three models mentioned above. The *Ontology Editor* is used to define the domain ontology, the task within a particular domain are described with the *Task Description Editor* that is supported by the *Summary Task Editor* that is used to define the task compositions of several sub-tasks to a larger task. The *Organization Editor* is used to define the organizational relationships among the agents and their mutual beliefs, the *Agent Definition Editor* is the tool to construct the agents with the architecture described earlier. The *Coordination Editor* is either used to select existing protocols from a protocol library or to define new protocols from scratch. A *Fact/Variable Editor* is provided to describe facts that hold in the problem domain using the ontology defined earlier. Furthermore, a *Constraint Editor* is used to specify the constraints among one or more preconditions of a single tasks, the precondition and effects of different tasks or the effects of a preceding task and the preconditions of a succeeding task in a summary tasks description. The *Code Generation Editor*, finally, is used to specify the code characteristics of the target language.

Besides this extensive collection of editors that allow for the easy definition of various aspects of the target system, the ZEUS tool-kit also provides a large

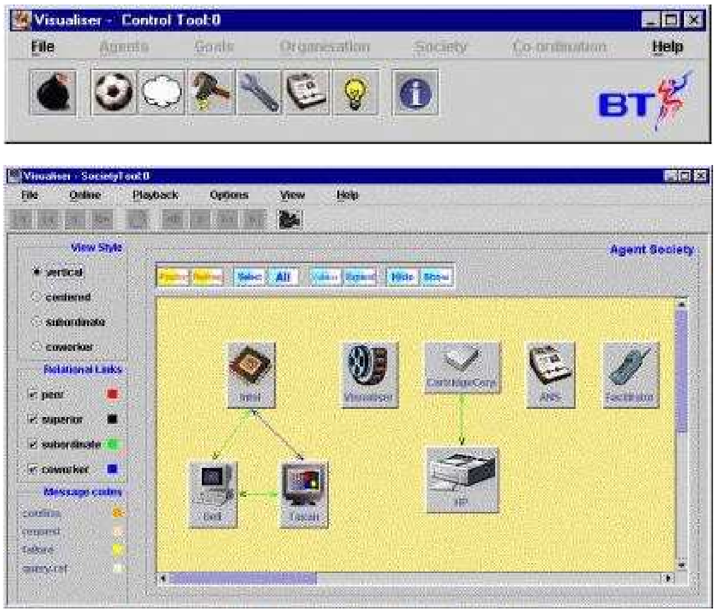


Figure 7.10: ZEUS Screenshot

collection of tools for the visualization and debugging of the application. The most important tool is the *visualizer* that is itself implemented as agent. The visualizer requests local information from the other agents and constructs a global view of the system that is presented to the designer or to the user. Additionally, the tool-kit provides a *society tool* that serves as the message monitor for the messages that are exchanged among the agents, a *report tool* that shows the current state of task decomposition and execution, the *micro tool* that is used to inspect the internal state of an agent, the *control tool* to remotely modify the internal state of an agent, the *statistics tool* the generates individual and society-wide statistics and, finally, the *video tool* that is used record and replay system runs. A screen shot of the ZEUS system that contains the control tool and the society tool is shown in Figure 7.10.

All in all, the freely available ZEUS tool-kit provides a comfortable and flexible framework for the development of multiagent applications. However, ZEUS is limited to task-oriented systems and it restricts the designer to a particular agent architecture that may not be suited for all kinds of applications.

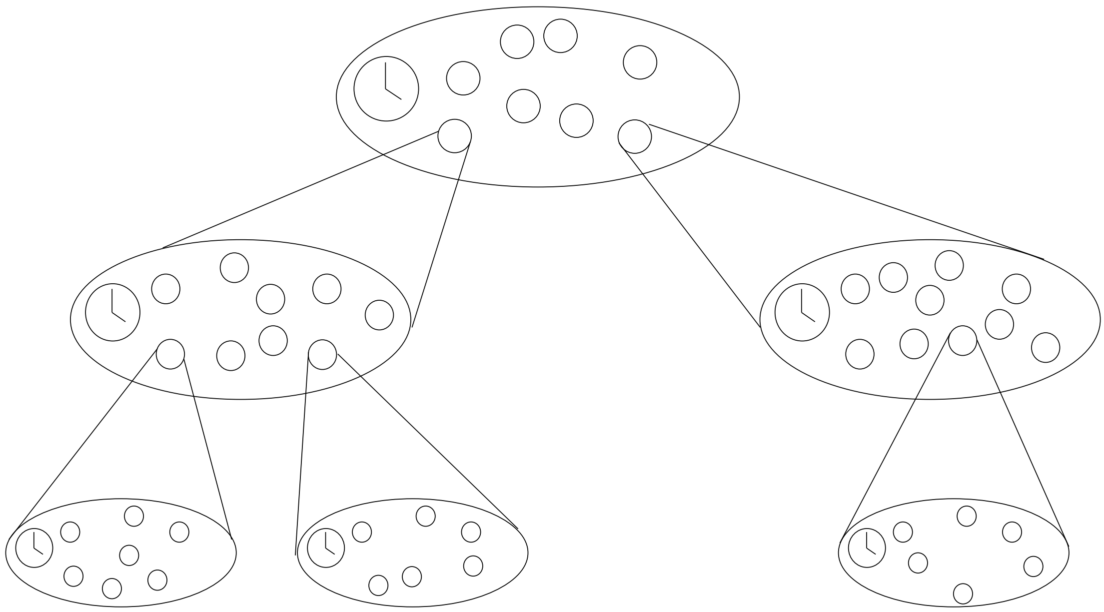


Figure 7.11: Swarm Hierarchy

7.2.3 Swarm

Swarm (HIEBELER, 1994), (MINAR ET AL., 1996), (BURKHART, 1997) is a domain-independent simulation toolkit that is based on a discrete event mechanism. The basic units of a Swarm simulation are *agents* that generate *events*, the events that are exchanged among agents is called a *schedule* of events. The structure that is built on top of the basic units consists of a collection of agents and their schedule of events and is referred to as a *swarm*. Swarms represent recursive structures as a swarm can be seen as a single agent within another swarm as shown in Figure 7.11. Each of the ellipses represents a swarm that consists of several agents (circles) and a schedule (indicated by the clock symbol). In a swarm, some of the agents are themselves swarms as shown in the figure.

The Swarm toolkit provides several libraries that allow the designer to specify the events that the agents can generate and the (hierarchical) structure of the swarms. The provided libraries are grouped in three classes.

The *simulation library* contains `swarmobject` classes that are used to describe the base objects of the simulations, the `activity` classes that are responsible for the scheduling and execution support and the `simtools` that are used to specify the type of simulations, eg. interactive or batch mode, etc.

The *software support library* consists of the `defobjs` and `collections` classes that provide general-purpose OO support such as common data-structures etc., the `random` classes that implement the random number support and the `tkobj` classes that hold then Tcl/Tk (OUSTERHOUT, 1994) interface for the visualization of the simulation.

The *model specific library*, finally, embody classes for special purpose applications such as the `space` classes for two-dimensional discrete lattices, the `ga` classes for genetic algorithms or the `neuro` classes for a variety of neural networks.

The major advantage of the Swarm tool-kit is that it is an extremely flexible approach that allows for the hierarchical modeling of multiagent simulations. The holonic approach supports reuse and the assemblance of applications out of already developed components. However, the focus of Swarm simulations is clearly on applications that consist of agents with very little reasoning and communication capabilities. Therefore, it is only of limited use for the analysis of applications with complex individual entities.

7.3 Summary

In this chapter, we have reviewed some development methods and tool-kits for multiagent applications. However, only a brief introduction to these sometimes very complex topics was provided, the interested reader may consider (IGLESIAS ET AL., 1998) and (LIND, 1999B) for further reading.

Each of the discussed methods was described and assessed according to a fixed scheme that allows for the direct comparison of the methods. However, an objective assessment of the methods is almost impossible as they were all applied to different, mostly unique problems. Still, the presented material should provide the reader with a general idea of what the individual methods are about, which aspects of the life cycle they can cover and in which context they can be used.

The tool-kits that have been presented provide development support for multiagent systems with different characteristics. Whereas the Swarm tool-kit focuses on applications with fine grained agents that turn up in large numbers, the SIF library is concerned more with virtual, co-habited worlds that need flexible interaction and visualization mechanisms. The ZEUS tool-kit is probably closest to an industrial tool-box as it provides a vast amount of editors, code generators and pre-defined interaction and problem-solving mechanisms. Thus, before building their own middle-ware, potential multiagent application developers should consider the tool-support that is available on the market.

CHAPTER 8

Conclusion

IN THE FINAL CHAPTER, we will review the requirements of the introduction and briefly evaluate the MASSIVE method along these dimensions. Then, we will outline the future directions of the research and development of the MASSIVE method.

8.1 Achievements

In th eintroduction, the basic properties that an ideal software development method for multiagent systems should possess were defined. We want a method that is *flexible*, *open*, *simple*, *scalable*, *supports learning and reuse*, *has little institutional overhead* and *relates to established software engineering concepts*.

In order to accomplish these goals, we have provided a development method that was built according to these fundamental requirements. The MASSIVE method consists of three major parts. The core of the MASSIVE method is a system of views that form the conceptual basis for a wide range of product models that are developed and refined throughout all software projects that are carried out according to the suggested process model. The main idea of the process model is an Iterative View Engineering approach that is itself based on Iterative Enhancement and Round-trip Engineering. The overall process model contains several small micro models that are used for individual tasks within the design process for a particular view. These micro models as well as the overall process model are not fixed for the entire lifetime of the project model, but they are subject to changes and refinements during the course of time. In order to preserve these adaptations and to make them accessible to others, the process model and the product model are both embedded into a larger organizational structure called the Experience Factory. The Experience Factory provides the formal framework for a permanent learning process that takes place over project boundaries and that eventually models the multiagent experience of organization in terms of specific product and process models for various domains.

The MASSIVE method implements the above stated requirements as follows. The MASSIVE method is *flexible* because it allows the designer to use and adapt products and processes on various levels of abstraction. As stated above, the macro development process consists of several micro processes for individual parts of the product model and this macro process as well as the micro processes can be adapted according to the needs of a particular project. The product model that has been introduced provides a generic base structure for a wide range of multiagent systems and can also be adapted according to project specific requirements.

The MASSIVE method is *open* in that it does not rely on a particular technology such as a programming language or a specific computational model. It has been demonstrated in the case studies that the method works with such different programming languages as Oz, Java or C++ and that it works for fully distributed applications as well as for threaded applications in a single process.

The method is sufficiently *simple* in that it is straightforward to apply and does not require intensive user training. A lot of the concepts used in the MASSIVE method are common sense and thus easily accessible for the developer. The product model is detailed enough to enable the developer to structure the target system without getting lost in fine grained details that are irrelevant in a particular context.

The MASSIVE method is *scalable* as it can be used by a single developer playing all roles as well as by a larger development team where the roles are assigned to actual persons.

Learning and reuse over project boundaries are supported by the concept of the Experience factory that is used to structure and package all sorts of experiences that are made during the course of a project. The resulting knowledge packages are selected for reuse in subsequent projects according to a case-based reasoning mechanism that provides the knowledge structures that are most suitable in a particular situation.

The MASSIVE method has only *little institutional overhead* because it can be introduced into a particular organization in parallel to the ongoing business processes. The Experience factory can be set up over a certain period of time without interfering with the current projects and it can be smoothly integrated step-by-step in the project execution process. The impact on the project organization are minimized by not forcing the team members to contribute to the experience acquisition process but rather having a separate organizational unit that is responsible for the analysis and evaluation of a project.

Finally, the method presented in this dissertation is *built upon standard software engineering* techniques and principles that have been developed over many years and demonstrated their validity in practical situations. The macro process of the MASSIVE method combines Iterative Refinement and Round-trip Engineering and puts them in the larger context of the Experience Factory. The suggested view-oriented product model is based on the ideas of aspect-oriented programming and allows the developer to model the system as a whole and viewing it from different angles.

In the next section, we will outline how research on the MASSIVE method could proceed in the future.

8.2 Further Work

The next step in the advancement of the development MASSIVE method is to set up some tool support for the activities that are performed in the course of a project. Interestingly, one of the most interesting topics in the design research community in recent years is multiagent technology to support the design process (LANDER, 1997) (PETRIE ET AL., 1998). The next step in the development of the MASSIVE method is to use the development method to built a multiagent system that supports the method itself!

But first of all, why are multiagent systems of particular interest to the design research community? According to (LANDER, 1997), *multiagent design systems (MADS)* provide the conceptual framework to model systems of flexible, sophisticated expert agents that collaborate in order to support a human project team in the course of the design process. The individual agents within the system can be personal assistant agents (wizards) that provide the necessary support for humans in the distributed design process.

The well-defined agent boundaries provide separable interfaces to the elements of the design process and support the locality of information because the internal specifications of a design element are not communicated over agent boundaries. Thus, each agent represents a particular design element or a design process and is responsible for keeping the other agents up to date about changes that are relevant to them while at the same time hiding changes that

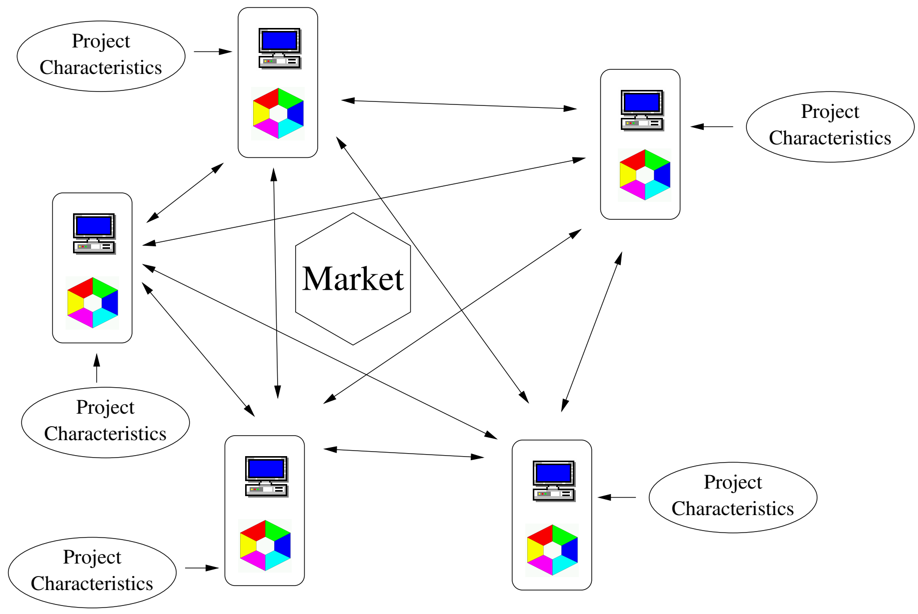


Figure 8.1: MADS View System Selection Process

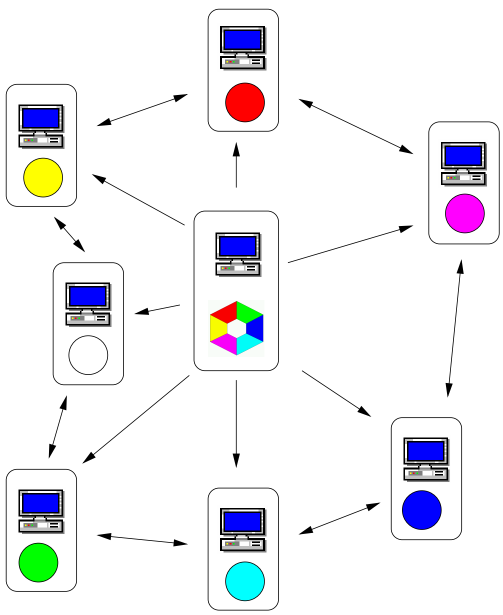


Figure 8.2: MADS Supported View Construction

have only local effects in order to limit the amount of information that is communicated.

To illustrate how a MADS for the MASSIVE method might look like, we will outline how the model selection process at the beginning of the project and the model construction process can be supported by a MADS. The first step in any project that is executed according to the MASSIVE development method is to find the most appropriate view system for the project. In Figure 8.1, we have several agents and each of those agents represents a particular view system. Now the user announces the project characteristics to the agents and they begin a negotiation process in order to find out which of them represents a model that is most suited for the project.

In a second step, the winning agent of the model selection process is then split up into several sub-agents each of which represents a particular view in the view system as shown in Figure 8.2. The sub-agents can be implemented as assistance agents that support the user in the construction of a particular view and that informs other sub-agents about relevant changes in the design.

Appendix A

Glossary

Agent An autonomous, pro-active, reactive entity with social abilities to interact with other entities.

Agent Architecture A structural model of the components that constitute an agent as well as the interconnections of these components together with a computational model that implements the basic capabilities of the agent.

Aspect-Oriented Programming A design technique that conceptualizes the idea of collecting several cross-cutting aspects of the software design in a single abstraction.

Deployment The process that covers all activities that are performed after a software system has been developed.

Environment Either the organizational or the runtime context of the target system.

Experience Factory A conceptual framework that supports systematic learning within an organization.

Feature A conceptual abstraction that can represent model entities such as design decisions or or physical entities such as components, devices or code fragments.

Feature Aggregation The process of grouping several features into a single abstraction that represents the entire group.

Feature Group The transitive refinement closure of a collection of features.

Feature Map A visualization of feature groups and/or views.

Feature Refinement The process of decomposing a single feature into several other features and therewith increasing the degree of detail of the model.

Implementation The manifestation of a model into the code of a particular programming language.

Interaction The mutual adaption of the behavior of agents.

Iterative View Engineering A software development model that combines Round-trip Engineering and Iterative Refinement.

Multiagent System Systems with a variable number of interacting, autonomous entities that communicate with each other using flexible, complex protocols.

Notation Is a language for communicating decisions that are not obvious or cannot be inferred from the code itself, that provides rich enough semantics sufficient to capture all important strategic and tactical decisions and offers a concrete form for humans to reason about decisions.

Performance Engineering A method to identify and reduce or eliminate performance problems during the software development cycle and the code has been designed and developed.

Process Model A formalization of the software design and implementation activities and of the products that are connected with these activities.

Product Model A representation of the characteristic features of a class of documents that constitute the the description of a software system.

PTA A test domain for the FIPA agent standard and for multiagent system design in general.

Role A logical grouping of functions that obeys the physical constraints of the operational environment of the target system.

Round-trip Engineering A software engineering process model that combines constructive code generation with analytic model refinement.

Shared Feature A feature that belongs to several views at the same time and that represents a dependency between these views.

Society A structured set of agents that agree on a minimal set of acceptable behaviors.

Social system A society that implements a closed functional context with respect to a common goal.

Software Development Method A combination of a notation, whose purpose is to provide a common means of expressing strategic and tactical decisions, ultimately manifesting themselves in a variety of artifacts and a process, responsible for specifying how and when certain artifacts should be produced.

System Architecture The fundamental structural attributes of a software system.

TCS A novel approach in freight transport scheduling on the railway.

Teamwork Approach A programming scheme for distributed search in large, unstructured search spaces.

UML A general purpose description language for software blueprints.

View A set of conceptually linked features that forms a projection of the complete model onto a particular aspect.

APPENDIX B

Example Protoz Specification for the Contract-Net Protocol

```
protocol ContractNet

  % channel for messages between manager and bidder

  channel
     Manager(Announce Grant Reject)
     Bidder(Bid NoBid Report)
  end


  % role manager; one instance is allowed within one
  % protocol execution

  role(1) Manager(agentList BidderList,
                  any Task, integer TimeoutValue)


    % channel between the protocol and the applicationProcedures
    % of the agent that has instantiated this role
    %
    % names and parameters of messages to applicationProc correspond
    % with names and parameters of the procedures
    %
    % messages from applicationProc are named as the procedures with
    % the word 'Result' in front

    channel
      protocol(ChooseBid(any Bid1, any Bid2)
               Inform(string Message)
               GetResult)

      applicationProc(ResultChooseBid)
    end



    %my states
    states
      WaitingForAnswer
      WaitingForReport
    end
```

```
declare

  %my variables
  message
    BestBid % the currently best bid
    Result
  end

  agent
    Contractor
  end

end


timeout
  Timeout1 TimeoutValue startOnEnter in WaitingForAnswer end
  Timeout2 TimeoutValue startOnEnter in WaitingForReport end
end

initialize to WaitingForAnswer
provided (length(BidderList) >= 1)
begin

  send Announce(Task) to Bidder.BidderList % send the message to all the
                                           % agents in the list

end


initialize to done
provided (length(BidderList) < 1)

% to less agents to announce to; there must be 2 or more
begin
  applicationProc(Inform("bidderList must have at least 2 elements"))
end


trans


  % transition 1

  from WaitingForAnswer to same
  when Bidder.BidderList % one of the agents in the list
  sends Bid
  provided (BestBid == nil)

  % this the first bid
  begin
     remove(sender BidderList)
    BestBid = currentMessage
  end


  % transition 2

  from WaitingForAnswer WaitingForReport to same
  when Bidder.BidderList sends Bid
  provided ((BestBid \= nil)  && (length(BidderList) > 1))

  %there was a bid before
  begin
    remove(sender BidderList)
    insert(CompareBids)    % insert macro
  end


  % transition 3

  from WaitingForAnswer to same
  when Bidder.BidderList sends NoBid
  provided (length(BidderList) > 1)  % not the last reply

  begin
    remove(sender BidderList)
  end
```

```
% transition 4

from WaitingForAnswer to WaitingForReport
when Bidder.BidderList sends Bid
provided (length(BidderList) == 1)

% last reply
begin
  remove(sender BidderList)
  insert(CompareBids)
  Contractor = getSender(BestBid)
  send Grant() to Bidder.Contractor
end


% transition 5

from WaitingForAnswer to WaitingForReport
when Bidder.BidderList sends NoBid
provided ((length(BidderList) == 1) &&
          (BestBid \= nil))

% last reply
begin
  remove(sender BidderList)
  Contractor = getSender(BestBid)
  send Grant() to Bidder.Contractor
end


% transition 6

from WaitingForAnswer to WaitingForReport
when system sends Timeout1
provided (BestBid \= nil)    % there is a bid
begin
  applicationProc(Inform("timeout in WaitingForAnswer, taking currently best bid"))
  Contractor = getSender(BestBid)
  send Grant() to Bidder.Contractor
end


% transition 7

from WaitingForAnswer to done
when system sends Timeout1
provided (BestBid == nil)    % there is no bid
begin
  applicationProc(Inform("timeout in WaitingForAnswer and no bid received"))
end


% transition 8

from WaitingForReport to done
when Bidder.Contractor sends Report
begin
  send GetResult(currentMessage.content) to applicationProc
end


% transition 9

from WaitingForReport to done
when system sends Timeout2
begin
  applicationProc(Inform("timeout in WaitingForReport and no report received"))
end


% transition 10

from WaitingForReport to same
when Bidder.BidderList sends Bid
begin
  send Exception("bid arrived to late") to Bidder.sender
end


% transition 11
```

```
    from WaitingForAnswer to done
    when Bidder.BidderList sends NoBid
    provided ((length(BidderList) == 1) &&
              (BestBid == nil))

    % last reply but currently no bid
    begin
      applicationProc(Inform("no bid received"))
    end


  end




  macro CompareBids
  begin

    Result = applicationProc(ChooseBid(BestBid.content currentMessage.content))

    if (Result.content == BestBid.content)  % the new bid is worse
    then
      send Reject() to Bidder.getSender(currentMessage)

    elseif (Result.content == currentMessage.content) % the new bid is better
    then
      send Reject() to Bidder.getSender(BestBid)
      BestBid = currentMessage

    else    % error
      applicationProc(Inform("wrong result of method chooseBid"))
      exit
    end
  end


end



% role bidder; multiple instances are allowed within
% one protocol execution

role(*) Bidder


  %my methods

  channel
    protocol(ComputeBid
             Execute
             Inform(string Message))

    applicationProc(ResultComputeBid ResultExecute)
  end



  %my states
  states
    Start ComputeBid WaitingForAnswer
    Execute
  end



  declare

    %my variables
    any MyTask end
    agent MyManager end

  end


  timeout
    Timeout1 12000 startOnEnter in WaitingForAnswer end
  end
```

```
    initialize to Start
    begin
    end

    trans

      % transition 1

      from Start to ComputeBid
      when Manager.anyAgent sends Announce
      begin
        MyManager = sender
        MyTask = currentMessage.content    % remember task
        send ComputeBid(MyTask) to applicationProc
      end


      % transition 2

      from ComputeBid to WaitingForAnswer
      when applicationProc sends ResultComputeBid
      provided (currentMessage.content \= nil)

      % content is not nil; send bid
      begin
        send Bid(currentMessage.content) to Manager.MyManager
      end


      % transition 3

      from ComputeBid to done
      when applicationProc sends ResultComputeBid
      provided (currentMessage.content == nil)

      % content is nil; send no bid
      begin
        send NoBid() to Manager.MyManager
      end


      % transition 4

      from WaitingForAnswer to done
      when Manager.MyManager sends Reject
      begin
        applicationProc(Inform("received reject"))
      end


      % transition 5

      from WaitingForAnswer to Execute
      when Manager.MyManager sends Grant
      begin
        send Execute(MyTask) to applicationProc
      end


      % transition 6

      from WaitingForAnswer to done
      when system sends Timeout1
      begin
        applicationProc(Inform("timeout and no reaction from manager"))
      end


      % transition 7

      from Execute to done
      when applicationProc sends ResultExecute
      begin
        send Report(currentMessage.content) to Manager.MyManager
      end
    end

  end

end
```

Bibliography

ADAMI, C. (1998). *Introduction to Artificial Life.* California Institute of Technology.

AVENHAUS, J. AND DENZINGER, J. (1993). Distributing equational theorem proving. In *Proceedings of the RTA'93*, number 690 in LNCS, Montreal.

AXELROD, R. (1984). *The Evolution of Cooperation.* Basic Books.

BACHEM, A., HOCHSTÄTTLER, W., AND MALICH, M. (1992). Simulated Trading: A New Approach for Solving Vehicle Routing Problems. Technical Report 92.125, Mathematisches Institut der Universität zu Köln.

BACHEM, A., HOCHSTÄTTLER, W., AND MALICH, M. (1993). The Simulated Trading Heuristic for Solving Vehicle Routing Problems. Technical Report 93.139, Mathematisches Institut der Universität zu Köln.

BAHRDT, H. P. (1994). *Schlüsselbegriffe der Soziologie.* C. H. Beck, München.

BALZERT, H. (1998a). *Lehrbuch der Software-Technik*, volume II. Spekrum Akademischer Verlag.

BALZERT, H. (1998b). *Lehrbuch der Software-Technik*, volume I. Spekrum Akademischer Verlag.

BARBUCEAN, M. AND FOX, M. S. (1995). The Architecture for an Agent Based Infrastructure for Agile Manufacturing. In *Proceedings of the Fithteenth International Joint Conference on Artificial Intelligence (IJCAI-95)",*.

BASILI, V. R. (1989). The Experience Factory: Packaging Software Experience. In *Proceedings of the 14th International Conference on Software Engineering.* NASA Goddard Space Flight Center.

BASILI, V. R. (1993). Applying the Goal/Question/Metric Paradigm in the Experience Factory. In *Proceedings of the 10th Annual CSR Workshop.*

BASILI, V. R., CALDIERA, G., AND ROMBACH, H. D. (1994). Experience Factory. In MARCINIAK, J. J., editor, *Encyclopedia of Software Engineering*, volume 1, pages 469–476. John Wiley & Sons.

BASILI, V. R. AND TURNER, A. J. (1975). Iterative Enhacement: A Practical Technique for Software Development. In *Proceedings of the First National Conference on Software Engineering*, pages 56–62. IEEE Computer Society Press.

BAYRISCHE LANDESREGIERUNG (1996). Bayerninfo. http://www.bayerninfo.de.

BECK, K. (1999). *eXtreme Programming eXplained: Embrace Change*. Addison-Wesley.

BODEN, M. A., editor (1996). *The Philosophy of Artificial Life*. Oxford readings in philosophy. Oxford University Press, Oxford.

BOEHM, B. W. (1988). A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21(5):61–72.

BOOCH, G. (1996). *Object Solutions: Managing the Object-Oriented Project*. Object-Orieneted Software Engineering. Addison-Wesley.

BOOCH, G., RUMBAUGH, J., AND JACOBSON, I. (1999). *The Unified Modeling Language User Guide*. Addison Wesley.

BRADSHAW, J. M., editor (1997). *Software Agents*. MIT Press.

BRAZIER, F., DUNIN-KEPLICZ, B., JENNINGS, N., AND TREUR, J. (1997). Desire: Modelling multi-agent systems in a compositional formal framework. *International Journal of Cooperative Information Systems*, 6. Special Issue on Formal Methods in Cooperative Information Systems: Multi-Agent Systems.

BRAZIER, F., JONKER, C., AND TREUR, J. (1998). Principles of compositional multi-agent system development. In CUENA, J., editor, *Proceedings of the IFIP'98 Conference on Information Technology and Knowledge Systems*. Chapman and Hall.

BRAZIER, F., VAN ECK, P., AND TREUR, J. (1996). Design of a modelling framework for multi-agent systems. In ALBRECHT, R. AND HERRE, H., editors, *Trends in Theoretical Informatics*, Schriftenreihe der Österreichischen Computer Gesellschaft. R. Oldenbourg Verlag.

BROOKS, JR., F. P. (1986). No silver bullet. In KUGLER, H.-J., editor, *Proceedings of the IFIP Tenth World Computing Conference*, pages 1069–76, Elsevier Science.

BROOKS, R. A. (1991). Intelligence without representation. *Artificial Intelligence*, 47:139–159.

BÜRCKERT, H.-J., FISCHER, K., AND VIERKE, G. (1998). Transportation scheduling with Holonic MAS, the TeleTruck approach. In *Proceedings of the PAAM98*.

BURKHART, R. (1997). Schedules of activity in the swarm simulation system. In *roceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '97)*.

BURMEISTER, B. (1996). Models and methodology for agent-oriented analysis and design. In FISCHER, K., editor, *Working Notes of the KI'96 Workshop on Agent-Oriented Programming and Distributed Systems*, number D-96-06 in DFKI Documents. DFKI.

BURMEISTER, B., HADDADI, A., AND SUNDERMEYER, K. (1995). Generic configurable cooperation protocols for multi-agent systems. In CASTELFRANCHI, C. AND MÜLLER, J.-P., editors, *From Reaction to Cognition — 5th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'93)*, volume 957 of *LNAI*, pages 157–171. Springer-Verlag.

BURT, A. (1998). Emotionally Intelligent Agents: The Outline of a Resource-Oriented Approach. In *Proceedings of the 1998 AAAI Fall Symposium Emotional and Intelligent: The Tangled Knot of Cognition*.

BUSSMANN, S. AND MÜLLER, H. J. (1993). A Communication Structure for Cooperating Agents. *Computers and AI*, I.

CAMMARATA, S., MCARTHUR, D., AND STEEB, R. (1983). Strategies of cooperation in distributed problem solving. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence (IJCAI-83)*.

CARLEY, K. M. (1999). Computational organizational theory. In *Multiagent Systems - A Modern Approach to Distributed Artificial Intelligence*. MIT Press.

CARROUÉ, L. (1997). La ruinease maladie du ≪ tout-routier ≫ — une europe des transports menacée d'embolie. *Le Monde Diplomatique*, pages 18–19.

COLLINS, J. AND NDUMU, D. (1998). The ZEUS Role Modelling Guide. Technical report, BT, Adastral Park, Martlesham Heath.

CONTE, S. D., DUNSMORE, H. E., AND CHEN, V. Y. (1996). *Software Engineering Metrics and Models.* The Benjamin/Cummings Publishing Company.

COOK, W. A. (1979). *Case Grammar: Development of the Matrix Model.* PhD thesis, Georgetown University, Washington.

COX, I. J. AND GEHANI, N. H. (1989). Exception Handling in Robotics. *IEEE Computing*, 22(3).

DEFENSE ADVANCED RESEARCH PROJECTS AGENCY (1981). Transmission Control Protocol.

DEMARCO, T. (1978). *Structured Analysis and System Design.* Englewood Cliffs:Yourdon Press.

DENZINGER, J. (1993). *Teamwork: Eine Methode zum Entwurf verteilter, wissensbasierter Theorembeweiser.* PhD thesis, Universität Kaiserslautern.

DENZINGER, J. (1994). The teamwork approach to distributed search. Technical report, University of Kaiserslautern.

DENZINGER, J. AND LIND, J. (1996). TWlib - a library for distributed search applications. In *Proceedings of the ICS96-AI*, pages 101–108, Kaohsiung.

DYER, M. (1992). *The Cleanroom Approach to Quality Software Development.* Wiley and Sons.

EDELMAN, G. M. (1987). *Neural Darwinism : the Theory of Neuronal Group Selection.* Basic Books.

ENGELMORE, R. AND MORGAN, T., editors (1988). *Blackboard Systems.* Addison-Wesley.

EPSTEIN, J. M. AND AXTELL, R. (1996). *Growing Artificial Societies : Social Science from the Bottom Up (Complex Adaptive Systems).* Brookings Institution Press.

EURESCOM (1999). MESSAGE: Methodology for Engineering Systems of Software AGEnts. Technical report, EURESCOM.

F., H., K., S., AND E., W. (1998). Establishing Experience Factories at Daimler-Benz: An Experience Report. In *Proceedings of the 20th International Conference on Software Engineering.*

FABEL, P. (1996). Increasing the flexibility of freight traffic - using modular train units as an example. In *Proceedings of the World Congress of Railway Research (WCRR)*, Colorado Springs; USA.

FININ, T. AND FRITZSON, R. (1994). KQML — a language and protocol for knowledge and information exchange. In *Proceedings of the 13th International Distributed Artificial Intelligence Workshop*, pages 127–136, Seattle, WA, USA.

FIPA (1996). *AgenTalk Reference Manual.* NTT Communication Science Laboratories and Ishida Laboratory, Department of Information Science, Kyoto University.

FIPA (1997). Fipa '97 specification parts 1–7, version 1.0. The Foundation for Intelligent Physical Agents.

FIPA (1998). Fipa '98 specification parts 1–13, version 1.0. The Foundation for Intelligent Physical Agents.

FISCHER, K. (1993). Rollenverteilung unter gleichberechtigten agenten. In *Verteilte Künstliche Intelligenz.* BI Wissenschaftsverlag.

FISCHER, K., KUHN, N., MÜLLER, H. J., MÜLLER, J. P., AND PISCHEL, M. (1993). Sophisticated and Distributed: The Transportation Domain. In *Proceedings of MAAMAW-93*, Neuchatel, CH.

FISCHER, K., KUHN, N., AND MÜLLER, J. P. (1994). Distributed, knowledge-based, reactive scheduling in the transportation domain. In *Proceedings of the Tenth IEEE Conference on Artificial Intelligence and Applications*, pages 47–53, San Antonio, Texas.

FISCHER, K. AND MÜLLER, H. J. (1995). Cooperative problem solving in the transportation domain. In DERIGS, U., editor, *Proceedings of the International Conference on Operations Research (OR'94).* Springer-Verlag.

FLOYD, C. (1983). A Systematic Look at Prototyping. In BUDDE, R., KUHLENKAMP, K., MATHIASSEN, L., AND ZÜLLINGHOVEN, H., editors, *Approaches to Prototyping.*

FONER, L. N. (1993). What's an Agent, Anyway? A Sociological Case Study. Agents Group, MIT Media Lab. Agents Memo 93-01.

FOWLER, M. (1999). *Refatoring – Improving the Design of Existsing Code.* Object Technology Series. Addison-Wesley.

FOX, M. (1981). An organizational view of distributed systems. *IEEE Trans. on Man, Systems and Cybernetics*, 11(1):70–80.

FRANKLIN, S. (1997). *Artificial Minds*. MIT Press.

FRANKLIN, S. AND GRAESSER, A. (1997). Is it an agent, or just a program?: A taxonomy for autonomous agents. In *Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages*.

FULBRIGHT, R. D. AND STEPHENS, L. M. (1994). Classification of multiagent systems. Technical Report ECE-LMS-94-06, University of South Carolina, Columbia, SC 29208.

FUNK, P., GERBER, C., LIND, J., AND SCHILLO, M. (1998). SIF: An agent-based simulation toolbox using the EMS paradigm. In *Proceedings of the 3rd International Congress of the Federation of EUROpean SIMulation Societies (EuroSim)*.

GALITZ, W. O. (1997). *The Essential Guide to User Interface Design*. Wiley Computer Publishing.

GARLAN, D. AND SHAW, M. (1993). An introduction to software architecture. In AMBRIOLA, V. AND TORTORA, G., editors, *Advances in Software Engineering and Knowledge Engineering*, volume I. World Scientific Publishing.

GARLAN, D. AND SHAW, M. (1994). An introduction to software architecture. Technical Report CMU-CS-94-166, Carnegie Mellon University: Software Engineering Institute.

GASSER, L. (1995). Introduction to multi-agent systems. In *Working Notes of Tutorial A at ICMAS95*, San Francisco, CA.

GENESERETH, M. AND NILSSON, N. (1987). *Logical Foundations of Artificial Intelligence*. Morgan Kaufman Publishers, Inc.

GENESERETH, M. R. AND FIKES, R. E. (1992). Knowlege interchange format, version 3.0, reference manual. Technical Report Logic-92-1, Logic Group, Computer Science Department, University of Stanford.

GEORGEFF, M. P. (1983). Communication and interaction in multi-agent planning. In *Proceedings of the Third National Conference on Artificial Intelligence (AAAI-83)*, pages 125 – 129.

GERBER, C. (1997). An agent society is more then a collection of agents. In *Workshop Notes of the AAAI Fall Symposium 1997 on Socially Intelligent Agents*.

GERBER, C., SIEKMANN, J., AND VIERKE, G. (1999a). Holonic multi-agent systems. Technical Report TR-99-01, DFKI.

GERBER, C., STEINER, D., AND BAUER, B. (1999b). Resource adaptation for a scalable agent society in the motiv-pta domain. In *Intelligent Software Agents for Communication Networks*. Springer.

GOODWIN, R. (1993). Formalizing properties of agents. Technical Report CMU–CS–93–159, School of Computer Science, Carnegie-Mellon Universit, Pittsburgh, PA.

HALL, R. (1999). *Agent-based Software Configuration and Deployment*. PhD thesis, University of Colorado.

HEISE, D. (1992). Computer assistance in qualitative sociology. *Social Science Computer Review*, 10:531–543.

HIEBELER, D. (1994). The swarm simulation system and individual-based modeling. In *Proceedings of the Intenational Conference on Advanced Technology for Natural Resource Management*.

HOLZMANN, G. J. (1991). *Design and Validation of Computer Protocols*. Prentice Hall.

HORN, E. AND REINKE, T. (1999). Musterarchitekturen und entwicklungsmethoden für Multiagentsysteme. *KI*.

HUHNS, M. N. AND SINGH, M. P., editors (1998). *Readings in Agents*. Morgan Kaufmann, San Francisco, California.

IGLESIAS, C., GARRIJO, M., AND GONZALEZ, J. (1998). A Survey of Agent-Oriented Methodologies. In MÜLLER, J. P., SINGH, M. P., AND RAO, A. S., editors, *Intelligent Agents V — Proceedings of the 1998 Workshop on Agent Theories, Architectures, and Languages (ATAL-98)*, volume 1555 of *LNAI*.

IGLESIAS, C., GARRIJO, M., GONZALEZ, J., AND VELASCO, J. R. (1997). Analysis and design of multiagent systems using MAS-CommonKADS. In SINGH, M. P., RAO, A., AND WOOLDRIDGE, M. J., editors, *Intelligent Agents IV: Agent Theories, Architectures and Languages*, number 1365 in LNAI.

ISHIKAWA, K. (1985). *What is Total Quality Control? The Japanese Way*. Englewood/Cliffs, New York.

JALOTE, P. (1997). *An Integrated Approach to Software Engineering*. Spinger, 2nd edition.

JENNINGS, N. R. (1999). Agent-Oriented Software Engineering. In IMAN, I., KODRATOFF, Y., EL-DESSOUKI, A., AND ALI, M., editors, *Proceedings of the 12th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, number 1611 in LNAI. Springer.

JENNINGS, N. R., SYCARA, K. P., AND WOOLDRIDGE, M. (1998). A roadmap of agent research and development. *Journal of Autonomous Agents and Multi-Agent Systems*, 1(1):7–36.

JENNINGS, N. R. AND WOOLDRIDGE, M. J., editors (1998). *Agent Technology : Foundations, Applications, and Markets.* Springer, Berlin.

JUNG, C. G. (1999). *Theory and Pratice of Hybrid Agents.* PhD thesis, Universität des Saarlandes.

KENDALL, E. A. (1998a). Agent Analysis and Design with Role Models. Technical report, British Telecom. Volume I: Overview.

KENDALL, E. A. (1998b). Agent Analysis and Design with Role Models. Technical report, British Telecom. Volume II: Role Models for Agent Enhanced Workflow and Business Process Management.

KEPHART, J. O., HOGG, T., AND HUBERMAN, B. A. (1989). Dynamics of Computational Ecosystems: Implications for DAI. In GASSER, L. AND HUHNS, H. M., editors, *Distributed Artificial Intelligence, Volume II.* Morgan Kaufmann Publishers, Inc., San Mateo, CA.

KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C. V., LOINGTIER, J.-M., AND IRVIN, J. (1997). Aspect-Oriented Programming. In *Proceddings of the European Conference on Object-Oriented Programming (ECOOP)*, number 1241 in LNCS. Springer-Verlag.

KINNY, D. AND GEORGEFF, M. (1996). Modelling and design of multi-agent systems. In J. P. MÜLLER, M. J. W. AND JENNINGS, N. R., editors, *Intelligent Agents III — Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages (ATAL-96)*, number 1193 in LNCS, pages 1–20. Springer-Verlag.

KNECHT, A. (1996). Gecco - Toolkit zur direkten Manipulation grafischer Objekte. Master's thesis, University of Kaiserslautern.

KÖGL, C. (1995). Verteilte Berechnung von Gröbnerbasen unter Verwendung des Teamwork-Paradigmas. Master's thesis, Universität Kaiserslautern.

KOLB, M. (1995). A cooperation language. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS'95)*, pages 233–238.

KOLODNER, J. (1993). *Case Based Reasoning.* Morgan Kaufmann.

KRACKE, R., SIEGMANN, J., VOGES, W., BOECKER, J., AND ZIRKLER, B. (1995). Systemgestaltung des Schienengüterverkehrs unter Einsatz der Strategie des Train-Coupling and -Sharing. Technical report, Universität Hannover. Studie im Auftrag der DB AG.

KRONENBURG, M. (1995). Hierarchisierung der Teamwork-Methode mittels Zerlegungsplanung. Master's thesis, Universität Kaiserslautern.

KUHN, N., MÜLLER, H. J., AND MÜLLER, J. P. (1994). Simulating cooperative transportation companies. In BIETHAHN, J., HUMMELTENBERG, W., SCHMIDT, B., AND WITTE, T., editors, *Simulation als betriebliche Entscheidungshilfe*, chapter 18, pages 263–264. Vieweg Verlag, Braunschweig/Wiesbaden.

KÜNZEL, K. (1997). Eine graphische Benutzerschnittstelle und Analyseumgebung für Systeme basierend auf dem Teamwork-Konzept. Master's thesis, Universität Kaiserslautern.

LANDER, S. E. (1997). Issues in Multiagent Design Systems. *IEEE Expert.*

LANG, J. AND STUART, D. B. (1998). A study of the applicability of existing exception-handling techniques to component-based real-time software technology. *ACM Transactions on Programming Languages and Systems*, 20(2).

LANGTON, C., editor (1989). *Artificial Life*, Redwood City, Calif. Addison-Wesley.

LAWLER, E. L. AND WOODS, D. E. (1966). Branch-and-Bound Methods: A survey. *Operations Research*, 14(4):699–719.

LEE, L., NWANA, H., NDUMU, D., AND DE WILDE, P. (1998). The stability, scalability and performance of multi-agent systems. *BT Technology Journal*, 16(3).

LEOPOLD, T. (1995). Verteilte Lösung des Travelling-Salesman-Problems durch TEAMWORK. Master's thesis, Universität Kaiserslautern.

LICHTER, H., SCHNEIDER-HUFSCHMIDT, M., AND ZÜLLIGHOVEN, H. (1994). Prototyping in industrial software projects – bridging the gap between theory and practice. *IEEE Transactions on Software Engineering*, 20(11):825–832.

LIND, J. (1996a). TWLib – A Generic Library for TEAMWORK Applications. Master's thesis, University of Kaiserslautern.

LIND, J. (1996b). *TWLib – Reference Manual.* University of Kaiserslautern.

LIND, J. (1998). The EMS Model. Technical Report TM-98-09, DFKI, Stuhlsatzenhausweg 3, D-66123 Saarbrücken.

LIND, J. (1999a). A Process Model for the Design of Multi-Agent Systems. Technical Report TM-99-03, DFKI, Stuhlsatzenhausweg 3, D-66123 Saarbrücken.

LIND, J. (1999b). A Review of Multiagent Systems Development Methods. Technical report, British Telecom, Adastral Park Labs, Martlesham Heath, Suffolk.

LIND, J. AND BÖCKER, J. (1999). Optimising the Train Coupling and -Sharing system with a multi-agent approach. In *Proceedings of the 11th Mini-EURO Conference on AI in Transportation Systems and Science,* Helsinki.

LIND, J., BÖCKER, J., AND ZIRKLER, B. (1999a). Optimising the Operation Management with a Multi-Agent Appraoch - Using TCS as an Example. In *Proceedings of the World Congress on Railway Research (WCRR),* Tokyo.

LIND, J. AND FISCHER, K. (1998). Transportation Scheduling and Simulation in a Railroad Scenario: A Multi-Agent Approach. Technical Report TM-98-05, DFKI, Stuhlsatzenhausweg 3, D-66123 Saarbrücken.

LIND, J., FISCHER, K., BÖCKER, J., AND ZIRKLER, B. (1999b). Transportation Scheduling and Simulation in a Railroad Scenario: A Multi-Agent Approach. In *Proceedings of the Fourth International Conference on the Practical Application of Intelligent Agents and Multi-Agents,* London.

LIND, J., JUNG, C. G., AND GERBER, C. (1999c). Learning and Adaptivity in Intelligent Real-Time Systems. In *Proceedings of the Third International Conference on Autonomous Agents (Agents'99).*

LINGER, R. C. AND TRAMMELL, C. J. (1996). Cleanroom software engineering reference model version 1.0. Technical Report CMU/SEI-96-TR-022, Carnegie Mellon University.

MACGREGOR, R. AND BATES, R. (1987). The LOOM Knowledge Representation Language. Technical Report ISI/RS-87-188, University of Southern California.

MAYFIELD, J., LABROU, Y., AND FININ, T. (1995). Evaluating KQML as an agent communication language. In WOOLDRIDGE, M., MÜLLER, J. P., AND TAMBE, M., editors, *Intelligent Agents — Proceedings of the 1995 Workshop on Agent Theories, Architectures, and Languages (ATAL-95)*, volume 1037 of *LNAI*, pages 347–360. Springer-Verlag.

MEHLHORN, K. AND NÄHER, S. (1999). *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press.

MILLS, H., DYER, M., AND LINGER, R. (1987). Cleanroom software engineering. *IEEE Software*, pages 19–24.

MINAR, N., BURKHART, R., LANGTON, C., AND ASKENAZI, M. (1996). The swarm simulation system: A toolkit for building multi-agent simulations. Technical Report 96-06-042, Santa Fe Institute.

MULLENDER, S. (1993). *Distributed Systems*. ACM Press.

MÜLLER, J. P. (1996a). Control Architectures for Autonomous and Interactin Agents: A Survey. In CAVEDON, L., RAO, A., AND WOBCKE, W., editors, *Intelligent Agent Systems: Theoratical and Practical Issues*, number 1209 in LNAI.

MÜLLER, J. P. (1996b). *The Design of Intelligent Agents: A Layered Approach*, volume 1177 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag.

MÜLLER, J. P. (1998). The Right Agent (Architecture) to do the Right Thing. In MÜLLER, J. P., SINGH, M. P., AND RAO, A. S., editors, *Intelligent Agents V — Proceedings of the 1998 Workshop on Agent Theories, Architectures, and Languages (ATAL-98)*, volume 1555 of *LNAI*.

NAUR, P. AND RANDELL, B., editors (1969). *Software Engineering. Report on a Conference sponsored by the NATO Science Commitee*, Garmisch, Germany, 7th to 11th October 1968. NATO, Scientific Affairs Division.

NDUMU, D. T., NWANA, H. S., LEE, L. C., AND COLLINS, J. C. (1999). Visualising and debugging distributed multi-agent systems. In *Proceedings of the 3rd International Conference on Autonomous Agents*.

NECHES, R., FIKES, R., FININ, T., GRUBER, R., PATIL, R., SENATOR, T., AND SWARTOUT, W. (1991). Eanbling Technology for Knowledge Sharing. *Ai magazine*, 12(3):36–56.

NODA, I. (1995). Soccer Server: A Simulator of Robocup. In *Proc. of AI symposium 1995*. JAPANESE SOCIETY FOR ARTIFICIAL INTELLIGENCE.

NWANA, H. S., NDUMU, D. T., LEE, L. C., AND COLLINS, J. C. (1999). ZEUS: A tool-kit for building distributed multi-agent systems. *Applied Artifical Intelligence Journal*, 13(1):129–186.

OBJECT MANAGEMENT GROUP (1999). CORBA: Common Object Request Broker Architecture and Specification, revision 2.3.

ODGERS, B., SHEPHERDSON, J., AND THOMPSON, S. (1999). Distributed workflow co-ordination by proactive software agents. In *Proceedings of the IJCAI-99 Workshop on Intelligent Workflow and Process Management*.

O'HARE, G. M. P. AND JENNINGS, N. R., editors (1996). *Foundations of Distributed Artificial Intelligence*. Wiley & Sons, New York.

OPDYKE, W. F. (1992). *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, Urbana-Champaign.

OUSTERHOUT, J. K. (1994). *Tcl and the Tk Toolkit*. Addison-Wesley Professional Computing.

PARNAS, D. (1996). Why Software Jewels are Rare. *IEEE Computer*, 29(2).

PARNAS, D. AND CLEMENTS, P. (1986). A Rational Design Process: How and Why to Fake It. *IEEE Transactions on Software Engineering*, SE-12(2).

PARUNAK, H. V. (1995). Case Grammar: A Linguistic Tool for Engineering Agent-Based Systems. Technical report, Industrial Technology Institute.

PARUNAK, H. V. (1997). 'Go to the Ant': Engineering Principles from Natural Agent Systems. *Annals of Operations Research*, 75:69–101.

PARUNAK, H. V. (1999). *Multiagent Systems - A Modern Approach to Distributed Artificial Intelligence*, chapter 9. MIT Press.

PARUNAK, H. V., SAUTER, J., AND CLARKE, S. (1997). Towards the specification and design of industrial synthetic ecosystems. In *Proceedings of the 1997 Workshop on Agent Theories, Architectures, and Languages*.

PERNICI, B. (1990). Object with Roles. In *Proceedings of the ACM/IEEE International Conference on Office Information Systems*, Boston.

PETRIE, C., GOLDMANN, S., AND RAQUET, A. (1998). Agent-Based Project Management. In *Artificial Intelligence Today*, number 1500 in LNAI. Springer.

PHILIPPS, S. (1998). Entwurf und Implementierung eines Systems zur Definition und Ausführung von Protokollen für Multi-Agentensystemen. Master's thesis, Fachhochschule Trier.

PHILIPPS, S. AND LIND, J. (1999). Ein System zur Definition und Ausführung von Protokollen für Multi-Agentensystemen. Technical Report RR-99-01, DFKI.

PITZ, W. (1993). Realisierung eines Systems zum verteilten, wissensbasierten Gleichheitsbeweisen mit Hilfe der Teamwork-Methode (in german). Master's thesis, Universität Kaiserslautern.

POORE, J. H. AND TRAMMELL, C. J. (1996). *Cleanroom Software Engineering: A Reader.* NCC Blackwell, Oxford.

PROGRAMMING SYSTEMS LAB (1999). The mozart programming system. University of the Saarland. http://www.mozart-oz.org.

RAO, A. S. AND GEORGEFF, M. (1995). BDI Agents: from theory to practice. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 312–319, San Francisco, CA.

RATIONAL SOFTWARE (1999a). Goals of the UML. http://www.rational.com.

RATIONAL SOFTWARE (1999b). Performace Engineering: A Practical Approach to Performance Improvement. Whitepaper. http://www.rational.com.

RAWSON, S. (1992). *Analysing Organisations.* The Macmillon press.

ROBINSON, J. (1965). A machine oriented logic based on the resolution principle. *Journal of ACM 12*, 1:23–41.

ROMBACH, H. D. (1994a). Vorlesungsscript Software Engineering I.

ROMBACH, H. D. (1994b). Vorlesungsscript Software Engineering II.

ROVATSOS, M. (1999). LAYLA – An InteRRaP Extension for Layerd Learning in Repeated Games. Master's thesis, University of the Saarland.

ROVATSOS, M. AND LIND, J. (1999). Learning cooperation in repeated games. In *Proceedings of the IJCAI-99 Workshop on Agents Learning About, From and With other Agents.*

ROYCE, W. W. (1970). Managing the development of large software systems: Concepts and techniques. In *WESCON Technical Papers, v. 14,* Los Angeles. WESCON. Reprinted in Proceedings of the Ninth International Conference on Software Engineering, 1987, pp. 328–338.

RUMBAUGH, J., JACOBSON, I., AND BOOCH, G. (1999). *The Unified Modeling Language Reference Manual.* Addision-Wesley.

RUSSELL, S. AND NORVIG, P. (1995). *Artificial Intelligence: A Modern Approach.* Prentice Hall.

RUSSELL, S. J. AND WEFALD, E. H. (1991). *Do the Right Thing : Studies in Limited Rationality.* MIT Press.

SAUNDERS, T. F., HOROWITZ, B. M., AND MLEZIV, M. L. (1996). A New Process for Acquiring Software Architecture. Mitre Corporation.

SCHILLO, M., LIND, J., FUNK, P., GERBER, C., AND JUNG, C. (1999). SIF - The Social Interaction Framework System Description and User's Guide to a Multi-Agent System Testbed. Technical Report TR-99-02, DFKI GmbH.

SEARLE, J. R. (1969). *Speech Acts.* Cambridge University Press.

SHAW, M. (1995). Patterns for Software Architectures. In COPLIEN, J. AND SCHMIDT, D., editors, *Pattern Languages of Program Design,* volume I.

SHAW, M. AND GARLAN, D. (1996). *Software Architecture : Perspectives on an Emerging Discipline.* Prentice Hall.

SHOHAM, Y. (1993). Agent-oriented programming. *Artificial Intelligence,* 60(1):51–92.

SIEMENS AG (1997). Verfahren und Anordnung zur Ermittlung einer Route von einem Startpunkt zu einem Zielpunkt. Deutsche Patentanmeldung 197 46 417.3. Donald Steiner, Jürgen Lind, Alastair Burt and Hartmut Dieterich.

SLOMAN, A. (1996). What sort of architecture is required for a human-like agent? In *Cognitive Modeling Workshop, AAAI96.* Invited Talk.

SMITH, C. U. (1997). Performance Engineering for Software Architectures. In *Proceedings of the Twenty-First Annual International Computer Software and Applications Conference (COMPSAC'97).* IEEE COMPUT. SOC.

SMITH, R. (1980). The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers.*

SMOLKA, G. (1995). The Oz programming model. In VAN LEEUWEN, J., editor, *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin.

SMULLYAN, R. (1968). *First-Order Logic*. Springer.

SO, Y. AND DURFEE, E. H. (1998). Designing Organizations for Computational Agents. In *Simulating Organizations*. MIT Press.

STALLMAN, R. M. AND FREE SOFTWARE FOUNDATION (1999). Emacs. http://www.gnu.org.

STEINER, D. (1992). MEKKA: Eine Entwicklungsumgebung zur Konstruktion kooperativer Anwendungen. In MÜLLER, J. AND STEINER, D., editors, *Kooperierende Agenten*, number D-92-24 in DFKi Document Series, pages 17–21. DFKI, Saarbrücken.

STROUSTRUP, B. (1987). *The C++ Programming Language*. Addison-Wesley, Massachusetts.

SUN MICROSYSTEMS (1999). The Java Programming System. http://java.sun.com.

SUNDERMEYER, K. (1993). Modellierung von agentensystemen. In *Verteilte Künstliche Intelligenz*. BI Wissenschaftsverlag.

TANENBAUM, A. S., editor (1988). *Computer Networks*. Prentice Hall, 2nd edition.

TEL, G. (1994). *Introduction to Distributed Algorithms*. Cambridge University Press.

THE DSDM CONSORTIUM (1998). DSDM method overview. http://www.dsdm.org/.

THE INTERNATIONAL ORGANIZATION FOR STANDARDIZATION (1997). IS-9074 (Information processing systems/Open systems interconnection): Estelle — a formal description technique based on an extended state transition model.

THE INTERNATIONAL ORGANIZATION FOR STANDARDIZATION (1998). Iso-35.100.05 multilayer application.

THE VRML CONSORTIUM (1997). *The Virtual Reality Modeling Language, ISO/IEC DIS 14772-1*. unknown.

VERLAGE, M. (1994). Multi-view modelling of software processes. In WARBOYS, B. C., editor, *Proceedings of the 3rd European Workshop on Software Process Technology*.

VOGES, W. AND MIERAU, U. (1997). Train Coupling & -Sharing. In *Proceedings of the World Congress of Railway Research (WCRR)*, Florence; Italy.

WEISS, G., editor (1999). *Multiagent Systems - A Modern Approach to Distributed Artificial Intelligence*. MIT Press.

WELLMAN, M. P. (1996). Market-Oriented Programming: Some Early Lessons. In CLEARWATER, S. H., editor, *Market-based Control*. World Scientific.

WERNER, E. (1988). A formal computational semantics and pragmatics of speech acts. In *Proceedings COLING–88*, pages 744–749.

WERNER, E. (1989). Cooperating agents: A unified theory of communication and social structure. In GASSER, L. AND HUHNS, M. N., editors, *Distributed Artificial Intelligence*, volume II.

WINDHOFF AG (1996). CargoSprinter. http://www.windhoff.de.

WOOLDRIDGE, M. (1997). Agent-based software engineering. *IEE Proceedings on Software Engineering*, 144(1):26–37.

WOOLDRIDGE, M., JENNINGS, N. R., AND KINNY, D. (1999). A methodology for agent-oriented analysis and design. In ETZIONI, O., MULLER, J. P., AND BRADSHAW, J., editors, *Agents '99: Proceedings of the Third International Conference on Autonomous Agents*.

WOOLDRIDGE, M. J. AND JENNINGS, N. R. (1998). Pitfalls of agent-oriented development. In *Proceedings 2nd International Conference on Autonomous Agents (Agents-98)*, pages 385–391, Minneapolis.