Michael Schumacher

Hot Topics

LNAI 2039

Objective Coordination in Multi-Agent System Engineering

Design and Implementation

Springer

Lecture Notes in Artificial Intelligence 2039

Subseries of Lecture Notes in Computer Science
Edited by J. G. Carbonell and J. Siekmann

Lecture Notes in Computer Science
Edited by G. Goos, J. Hartmanis and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Singapore
Tokyo

Michael Schumacher

Objective Coordination in Multi-Agent System Engineering

Design and Implementation

Springer

Series Editors

Jaime G. Carbonell, Carnegie Mellon University, Pittsburgh, PA, USA
Jörg Siekmann, University of Saarland, Saabrücken, Germany

Author

Michael Schumacher
Berninastrasse 85, 8057 Zürich, Switzerland
E-mail: ms@ymail.ch

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Schumacher, Michael:
Objective coordination in multi-agent system engineering : design and implementation / Michael Schumacher. - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 2001
(Lecture notes in computer science ; Vol. 2039 : Lecture notes in artificial intelligence)
ISBN 3-540-41982-9

CR Subject Classification (1998): I.2.11, C.2.4, D.1.3, D.2.12, I.2.9

ISBN 3-540-41982-9 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

Printed in Germany

Typesetting: Camera-ready by author, data conversion by Christian Grosche, Hamburg
Printed on acid-free paper SPIN: 10782507 06/3142 5 4 3 2 1 0

Foreword

This book addresses the engineering of Multi-Agent Systems (MAS) based on suitably-defined coordination models and languages. Its contribution is twofold: on the one hand, the theoretical part defines a new coordination model (ECM) for designing MAS and proposes two corresponding coordination languages (STL++ and Agent&Co) for implementing designed systems in distributed environments; on the other hand, the experimental part presents a software prototype by starting off the existing Pt-pvm (a distributed threads environment developed by Oliver Krone, a former PhD student of our PAI research group) and validates the theoretical part with two case studies. Definitely one of the main result is that the ECM model and its explicit distinction of objective and subjective coordination define an original and comprehensive framework for modeling, designing, and implementing interactions in complex MAS.

This work has been carried out within the framework of the Parallelism and Artificial Intelligence research group (PAI) at the University of Fribourg, Switzerland (`http://www-iiuf.unifr.ch/pai`). The PAI group originated from a synergy between the fields of Distributed Computing and Artificial Intelligence. Since 1986, it has been involved in massively distributed programming methodologies, and since 1995 in decentralized control strategies as far as they are suited for providing adaptive capabilities in network computing. Research topics of the group follow an engineering trend, which can be characterized by:

- The abandonment of centralized control and stringent hierarchical data structures in favor of decentralized control strategies based on interactions through influences. These strategies, which require autonomous components, lead to solutions, which are more flexible, more tolerant to perturbations, and which support the emergence of new properties;
- A bio-inspired approach, which draws its models from neuro-biology and the study of animal societies and participates in the concept of embodiment of intelligence now surfacing in AI;

Between 1995 and 1999, PAI's research fitted inside the AXE project (AXE is a compressed acronym for "Autonomy and Coordination Supports Evolution") whose aim was to devise a self-organization and a coordination

theory for massively distributed systems. AXE's main acquisitions concern (i) the development of coordination platforms for distributed applications; (ii) the definition of models for autonomous, adaptive, and evolutive agents; (iii) the design of coordination and evolution models, namely collective intelligence strategies. Since 1998, human computer interaction aspects are also part of PAI's research, in the measure that universal networking and ubiquitous computing formulate special requirements in this field. With AXE's termination at the beginning of 2000, this domain sets the new focus adopted by PAI inside the new WELCOME framework, building on the know-how acquired on autonomous systems and coordination.

It was a pleasure for me to work with Michael Schumacher. His multidisciplinary research results have been well received in the MAS and coordination communities. The modeling, design, and implementation of MAS, the ECM coordination model, and the STL++ and AGENT&CO coordination languages, which are the central parts of his PhD thesis described in this book, are now used in our projects presently under development in WELCOME.

Finally, I warmly acknowledge the financial support of the University of Fribourg, the Swiss National Science Foundation, and the European Programme of R&D for having granted PAI's research projects for the past fourteen years.

December 2000

Béat Hirsbrunner
Leader of the PAI research group

Preface

This book, situated at the intersection of behavior-based artificial intelligence and concurrent and distributed computing, defines programming paradigms to support the design and the concurrent and distributed implementation of *Multi-Agent Systems* (MAS) that simulate collective robotics applications. We analyze the research that has tried to fill the gap between agent theory and applications, observing that the proposed methodologies, languages, and tools are mostly concentrated on intra-agent aspects. In contrast to those approaches, we propose that the modeling of MAS should be a bottom-up and interaction-oriented process, grouping existing autonomous agents and describing how they interact, thus managing the *coordination* of these agents. To that aim, we distinguish *objective* coordination, which handles inter-agent dependencies (the organization of the environment and the agent interactions), and *subjective* coordination, which handles intra-agent dependencies often involving mentalistic categories. We then promote a methodology that focuses the modeling of MAS on objective coordination, and we propose the use of *coordination models* and corresponding *languages* (from the fields of concurrent and distributed computing, programming languages, and software engineering) in order to respectively support the design phase of a MAS and allow its implementation on a concurrent and distributed architecture.

After reviewing coordination models and languages, we examine the prerequisites that a coordination model and language should include in order to support our target MAS. On this basis, a general coordination model named ECM and a corresponding object-oriented coordination language named STL++ are presented. ECM/STL++ uses an encapsulation mechanism as its primary abstraction, offering structured separate name spaces which can be hierarchically organized. Agents communicate anonymously within and/or across name spaces through connections, which are established by the matching of the communication interfaces of the participating agents. Three basic communication paradigms are supported, namely point-to-point stream, group, and blackboard communication. Furthermore, an event mechanism is introduced for supporting dynamicity by reacting to state changes of the communication interfaces.

The use of ECM/STL++ is illustrated by the simulation of a particular collective robotics application and of the automation of a trading system.

Acknowledgements

This book compiles four years of research work that I carried out as a member of the Parallelism and Artificial Intelligence group of the Computer Science Department at the University of Fribourg in Switzerland, working on a PhD thesis funded by the Swiss National Science Foundation. I benefited enormously from the knowledge, the expertise, and the support of many people.

My special thanks go to: Béat Hirsbrunner who gave me the opportunity to work in a highly qualified research group and who has guided my research activity during the past years; Rolf Ingold, Andrea Omicini, and George Papadopoulos, for accepting the invitation to act as jury members; Fabrice Chantemargue and Oliver Krone for the numerous discussions, ideas, criticisms, and also for their constant help and enthusiasm; Antony Robert, Valerio Scarani, Michele Schumacher, and Robert Van Kommer for reviewing previous versions of this document; Christian Wettstein for implementing the first version of STL++; Ivan Doitchinov for implementing AGENT&CO; all colleagues in the PAI group for creating a stimulating research environment; and last but not least, my family and my friends for their constant support and encouragement.

January 2001 *Michael Schumacher*

Contents

1. Introduction

From its beginning, *Artificial Intelligence (AI)* has tried to synthesize intelligent behavior by reducing it to artifacts. Two families of approaches have, however, appeared [Zie97].

On one hand, *Knowledge-Based AI* (also named *Classical AI* or *Top-Down AI*, closely related to the classical view of cognitive sciences) has developed methodologies for manipulating internal representation models of the world, using notions like knowledge or information. This trend has focused on building *expert systems* (see [FMN88] for an overview) that own and manage 'knowledge' in a sufficiently narrow problem domain. Thus, research in this field has concentrated on knowledge representation and knowledge engineering. Several systems have shown good performance in domains like medical diagnosis and theorem proving.

On the other hand, *Behavior-Based AI* (also named *Bottom-Up AI*) considers interaction with an environment as an essential feature for intelligent behavior. Research in this area has studied systems that have 'life'-like attributes, in particular *autonomous agents* that persist within an environment. Autonomous agents are therefore defined as embodied systems that are designed to fulfill internal goals by their own actions in continuous long-term interaction with the environment in which they are situated [Zie97], [Bee95]. Autonomous agents possess two essential properties: *autonomy* and *embodiment*. Autonomy basically means that an agent acts on its own, without being driven by another agent (possibly a human): it has its own control over its actions and its internal state. Embodiment refers to the fact that an autonomous agent has a "body" that delineates it from its environment in which the agent is situated. It is on this environment that an agent senses and acts through its own specific means (sensors and effectors). The degrees of embodiment define several autonomous agent types: physical embodied agents have a physical body, sensors and effectors (they are robots); simulated embodied agents roam in a simulated physical environment; and software agents lack a body but persist and interact in a complex software environment.

Interest has rapidly shifted from the single agent case towards the multi-agent case, in which several agents share a common environment and interact with one another. This means that an agent participates in a society of agents, a so-called *multi-agent system*. These systems result therefore from the or-

M. Schumacher: Objective Coordination in MAS Engineering, LNAI 2039, pp. 1–5, 2001.

ganization of multiple agents within an environment, whereby the state of each agent individually changes as a result of its sensing and its own behavior rules. As no global control is applied to the participating agents, every agent has a subjective view of the evolution of the world. Multi-agent systems are therefore interested in an interaction-centered perspective. Dealing with interactions leads naturally to the concept of *emergence* of behavior, which essentially means that a system's behavior can only be specified using descriptive categories which are not to be used to describe the behavior of the constituent components (the agents) [For90].

This book is situated at the intersection of the target research areas of the PAI group[1], namely behavior-based artificial intelligence, and concurrent and distributed computing. Actually, its goal is to define programming paradigms to support the design and the implementation of a specific class of multi-agent systems composed of simulated embodied agents: it is concerned with the implementation of multi-agent simulations of collective robotics systems [BHD94], [MM95], [Mat95], [CH99], in which an emergent global behavior is achieved by very simple agent local rules and interactions restricted to the result of each one's work in the environment. Furthermore, the implementation of this class of multi-agent systems is intended to be realized on a specific underlying platform: a concurrent and distributed architecture.

This research is, therefore, concerned with a software engineering view of multi-agent systems: how can practical multi-agent systems be modeled, designed and implemented, and this on a concurrent and distributed architecture?

If one analyzes the research that has tried to fill the gap, often encountered, between agent theory and agent applications, he will observe several proposals for agent languages and tools for facilitating the design and the implementation of multi-agent systems, based upon generic agent architectures [WJ95]. Furthermore, he will see that this necessity has been recently recognized by various authors who advocate the study of methodological foundations of agent-based systems [FMS+97], [WJK99]. However, as noted in [COZ99b], the proposed agent languages, tools and methodologies are mostly concentrated on intra-agent aspects and neglect the interaction setup of the system: they generally define multi-agent systems in terms of the internal agent structures, usually based on mentalistic notions as does the BDI model (belief, desire and intention) [GR95].

In this book, we maintain, as does other recent research such as [Bur96], [Sin98b], [WJK99], [COZ99b], that the interaction setup should be clearly integrated in the analysis phase of a multi-agent system. The modeling of a multi-agent system should be a bottom-up and interaction-oriented process: the system should be constructed by grouping existing autonomous agents

[1] Parallelism and Artificial Intelligence Group of the Computer Science department at the University of Fribourg in Switzerland.

and by describing how they interact, thus by managing the *coordination* of these agents. But, as coordination is defined as *managing dependencies between activities* [MC94a], this kind of modeling is only possible if one identifies which dependencies exist within a multi-agent system. In this work, we therefore propose to distinguish between: i) *objective* dependencies, which refer to inter-agent dependencies, namely the configuration of the system in terms of the basic interaction means, agent generation/destruction and organization of the environment; and ii) *subjective* dependencies, which refer to intra-agent dependencies often involving mentalistic categories. Correspondingly, we maintain that two types of coordination exist in a multi-agent system: *objective* and *subjective* coordination. Not differentiating these two levels of coordination complicates a design and a subsequent implementation, because this leads to multi-agent systems that actually describe objective coordination with subjective coordination means, i.e. by using intra-agent aspects for describing system configurations.

Therefore, a multi-agent system modeling benefits by clearly identifying which objective dependencies are present and how they are handled. In other words this modeling is facilitated if one specifies how the environment is organized and how agents interact. It is only on this basis that one can appropriately design and implement a multi-agent system; and, like a few recent works such as [COZ99a] and [COZ99b], we maintain that this design and implementation can be done with an appropriate coordination model and language.

Indeed, research from concurrent and distributed computing, programming languages, and software engineering has developed several *coordination models and languages* [CG92a], [PA98c]. They are suitable precisely for supporting the design and the implementation of objective coordination in multi-agent systems, because they define abstractions dealing with the management of the interactions between activities and the organization of the activity space. Furthermore, the use of a coordination model and language is even more important if a multi-agent system runs on a distributed and concurrent architecture, which should be the natural substrate for its execution. In fact, coordination models and languages have a strong expertise in concurrent and distributed computing: they offer means to deal with the consequences of the concurrency and the spatial distribution of the supporting processes. This need for coordination models and languages is especially important for our target class of multi-agent systems, in which the organization of the environment does not reflect the organization of the underlying execution platform.

The contribution of this book can be summarized as follows:

– We promote a focus upon objective coordination issues when modeling a multi-agent system and present a general multi-agent system model for our target applications.

- We advocate and motivate the use of coordination models for designing objective coordination in multi-agent systems, and we present a concrete coordination model named ECM for sustaining the design of our specific class of systems.
- We advocate and motivate the use of coordination languages for implementing designed multi-agent systems onto a concurrent and distributed architecture, and we present a concrete coordination language named STL++ for implementing our target class of systems.
- We show an implementation in STL++ of a concrete application of our target multi-agent systems.

This work is therefore concerned with the organizational structure or architecture of multi-agent systems and does not focus on intra-agent aspects such as agent tasks and subjective coordination. As such, our proposed coordination language STL++ is not a complete multi-agent system language.

The book is organized in three parts. Part I positions our research in the fields of multi-agent systems (chapter 2) and coordination models and languages (chapter 3). Part II presents the ECM coordination model and its instances (chapters 4 to 7). Finally, part III is devoted to two case studies (chapters 8 and 9). Hereafter, we shortly describe the content of each chapter.

In chapter 2, we review the notions of multi-agent systems and autonomous agents by showing their essential characteristics. We then tackle the problematic of modeling a multi-agent system by distinguishing between subjective and objective coordination. Next we present our target class of multi-agent systems, and introduce a generic model and a peculiar application of it as have been defined in the PAI group. Finally we discuss the implementation of multi-agent systems and advocate the use of coordination models and languages for supporting the design and the implementation of these systems.

After having introduced coordination theory, chapter 3 reviews research in coordination models and languages by analyzing their motivation and key elements and by presenting an established taxonomy that distinguishes between data-driven and process-oriented models. Each family of models is introduced by its main representative and further illustrated by several other models. We then discuss various hybrid coordination models and conclude by presenting the prerequisites that the coordination model and language presented in this book should include in order to support our target multi-agent systems.

In chapter 4, we present the ECM model of coordination that we have defined together with colleagues from the PAI group. The model uses an encapsulation mechanism as its primary abstraction (referred to as blops), offering structured separate spaces which can be hierarchically organized. Within them, active entities communicate anonymously within and/or across blops through connections established by the matching of the communication interfaces (the so-called *ports*) of these entities. Three basic communication

paradigms are supported, namely point-to-point, group and blackboard communication. Furthermore, an event mechanism is introduced for supporting dynamicity by reacting to port state changes. Three instances of ECM are presented in the subsequent chapters: STL, STL++ and AGENT&CO.

In chapter 5, the STL coordination language, which has been the subject of another research work, is reviewed. STL is implemented as a separate language (supported by a compiler) that must be used in conjunction with a computation language that implements the coordinated processes. STL is applied to multi-threaded applications in the context of network-based concurrent computing.

In chapter 6, we present STL++, our main coordination language. STL++ is an object-oriented instance of ECM that follows the prerequisites of chapter 3 for our target class of multi-agent systems. It has a single language approach, i.e. the language is implemented as a library extended by few macros. The object-oriented integration is a target design issue: every abstraction of ECM is described by a class, and the primitives on these abstractions become member functions except some creation macros. Furthermore, STL++ emphasizes dynamicity and uniformity in the realization of the ECM constructs. In this chapter, STL++ is introduced step by step. The use of the coordination language is illustrated with a classical tutorial example. Finally, after having presented our implementation of STL++, we conclude with a discussion.

In chapter 7, we present AGENT&CO, another instance of ECM. This new language binding tries to enhance STL++ in two ways. Firstly, AGENT&CO slightly extends ECM in order to support mobility of active entities over blops. Secondly, the language is implemented in JAVA over an object request broker architecture in order to enhance portability.

In chapter 8 and 9, the implementation of two case studies are respectively discussed. The first is a realization of the peculiar application of our generic multi-agent system model presented in chapter 2. The second example is the implementation of the simulation of a trading system. Although this latter does not belong to our target class of systems, it shows that STL++ can also be applied to other multi-agent systems.

Finally, before giving a detailed description of the core STL++ interfaces in appendix A, chapter 10 concludes the book and outlines future work.

2. Multi-Agent Systems

2.1 Introduction

The interest for multi-agent systems (MASs) has grown increasingly in the last years. These systems are beginning to be used in a great variety of applications such as air traffic control, process control, manufacturing, electronic commerce, patient monitoring, or games. This appeal is due to the fact that multi-agent systems present very attractive means of more naturally understanding, designing and implementing several classes of complex distributed and concurrent software. This growing attention to multi-agent technology has been even more accentuated with the increase of internet computing, which integrates an underlying infrastructure presenting a space organization in which autonomous agents roam and interact with one another.

Nevertheless, as a consequence of their popularity, the terms "*autonomous agents*" and "*multi-agent systems*" are often misused. It is indeed not rare to find software described in these terms, although it is not necessarily correlated with multi-agent technology. Therefore, we have to clarify what autonomous agents and multi-agent systems really are, how they can be modeled, designed and implemented.

This chapter is organized as follows. In sections 2.2 and 2.3, we present what are autonomous agents and multi-agent systems. On this basis, we discuss in section 2.4 MAS modeling by proposing an explicit integration of the interaction setup of a MAS and by sustaining a distinction between subjective and objective coordination. Then, section 2.5 exposes our target class of systems. Finally section 2.6 discusses how practical MASs can be built and concludes by maintaining the use of coordination models and languages for the design and the implementation of MASs.

2.2 What Is an Autonomous Agent?

The debate on what constitutes an autonomous agent is still under way. As a large range of agent types exist, it is obvious that a lot of descriptions have been proposed, without, however, reaching a commonly accepted definition. This section nevertheless sketches the fundamental characteristics of

M. Schumacher: Objective Coordination in MAS Engineering, LNAI 2039, pp. 9–32, 2001.

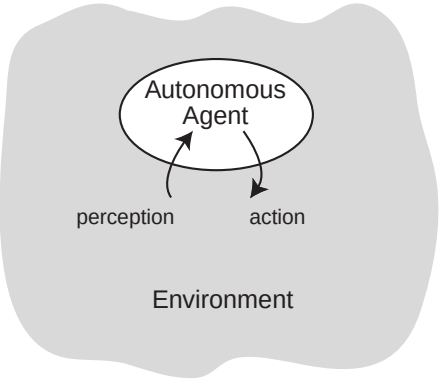

Fig. 2.1. An autonomous agent and its environment.

what most research understands with an autonomous agent by presenting the definition we adhere to and by discussing a wide-spread definition.

2.2.1 Definitions

Wishing to avoid the formulation of a new definition and being conscious of the difficulty of the task due to the wide variety of existing agent paradigms (see for instance [Mae95]. [HR95]. [BT96]. [JSW98]), this section proposes to adopt a proposal of Franklin and Graesser [FG96].

After having analyzed several agent proposals, Franklin and Graesser give a definition, which is similar to that of Beer [Bee95]. They explain the essence of an autonomous agent as follows:

> *An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future.*

This definition is further explained:

> *Each agent is situated in, and is part of some environment. Each senses its environment and acts autonomously upon it. No other entity is required to feed its input, or to interpret and use its output. Each acts in pursuit of it's own agenda. (...) Each acts so that its current actions may effect its later sensing, that is its actions effect its environment. Finally, each acts continually over some period of time.*

This definition shows that the notion of *situatedness* within an environment is an essential concept: the agent can sense its surrounding (receive sensory input) and act upon it so as to change it. Furthermore, the response

of the agent to the environment is done in a timely fashion. Figure 2.1 gives a schematic representation of those ideas.

In the literature, it is not rare to find descriptions of what an autonomous agent is, as proposed by Wooldridge and Jennings in [WJ95]. They define an autonomous agent as:

> *a hardware or (more usually) software-based computer system that enjoys the following properties:*
> – *autonomy: agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state;*
> – *social ability: agents interact with other agents (and possibly humans) via some kind of agent-communication language;*
> – *reactivity: agents perceive their environment (...), and respond in timely fashion to changes that occur in it;*
> – *pro-activeness: agents do not simply act in response to their environment, they are able to exhibit goal-directed behaviour by taking the initiative.*

This definition is not as general as the precedent, because it requires agents to interact using an agent communication language (ACL) [LFP99]. But there exist several types of autonomous agents that do not need such ACLs and only interact through influences in the environment (see section 2.5). Nevertheless, this definition is considered by Wooldridge and Jennings as a *weak* notion of agency, in contrast with a *strong* notion of agency that uses mentalistic notions applied to humans (see section 2.2.3) and that represents one of several existing generic agent architectures. Before presenting these architectures in section 2.2.3, we discuss more in detail two characteristics of autonomous agents, namely autonomy and embodiment.

2.2.2 Autonomy and Embodiment

The definitions discussed in section 2.2.1 clearly refer to a basic notion of autonomy: an agent is considered autonomous if it acts for its proper agenda by choosing, by itself, its actions. But, intuitively, we see that there may exist different degrees of autonomy, including stronger ones. In fact Ziemke distinguishes two families of autonomy [Zie97]: *operational* and *behavioral* autonomy. Operational autonomy is the capacity to operate *without human intervention* [Pfe96]. This is the simplest and basic meaning. The behavioral autonomy, however, has a stronger definition: it can be considered as having its *own capacity to form and adapt its principles of behaviour* [Ste95]. Note that this second autonomy is not a necessary characteristics for the agent definition adopted.

But autonomy also means that each agent possesses its proper temporality in the sense that it runs concurrently to other agents without being synchro-

nized to them. This requirement should especially be taken in consideration when implementing agent-based systems.

Strongly coupled to the notion of autonomy is the fact that an autonomous agent is an embodied system, i.e. the fact that an autonomous agent has a "body" that delimits and isolates the agent from its environment. Different embodiments yield to different kinds of agents [Zie97]:

- *Physically embodied agents* subsist in a physical environment: their interaction with the environment goes through sensory perception and motor control. They are typically robots, or even living agents.
- Several agents use a *simulated embodiment*: simulated physical agents are roaming and acting in a simulated world, thus physical embodied agents and a realistic environment are only simulated.
  It is indeed very useful to realize simulations of cost and time intensive experiments before one completes a final real (physical) implementation. One has, for instance, the possibility to experiment several different control algorithms for evaluating them. Realizing statistical studies is also much easier with simulated embodied autonomous agents, because implementations are more efficient and faster. In certain cases, a lack of physical resources (physically embodied) also motivates the implementation of simulated embodied agents.
- Finally, *software agents* [Nwa96], [Sho99] directly interact with software environments. They lack of a body, sensors and motor, but are nevertheless situated within an often complex software environment. Famous examples are the *interface agents*, which are personal assistants of a user, and the *information agents*, which find on the WWW information specifically interesting a user [Mae94].

The second and third class of agents are obviously computational agents. The presented taxonomy is summarized in figure 2.2. As we shall see in section 2.5, our target class of MASs comprises simulated embodied agents.

2.2.3 Generic Agent Architectures

Autonomous agents are systems in continuous long-term interaction with an environment in which their are situated. This constitutes a common basis for all autonomous agents. But based on so-called agent theories [WJ95], different *generic agent architectures* have appeared, differing from one another in their view of the intra-agent aspects. They can be grouped in *mental*, *reactive* and *hybrid agents* (for an overview, see for instance [WJ95] or [Oss99]).

Mental Agents. Mental agents, also named *rational* or *deliberative agents*, are viewed as intentional systems [Den87], using mentalistic notions applied to humans. The agent has an explicit symbolic model of the world in which it lives. It uses sensor data in order to update its model of the environment. A planner reasons on the world model and decides which actions to realize.

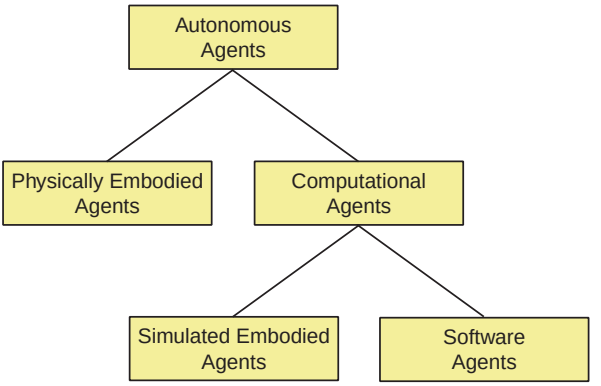

Fig. 2.2. A basic agent taxonomy in relation to the notion of embodiment.

In contrast to the *weak* notion (see section 2.2), Wooldridge and Jennings consider mental agency as a *strong* notion of agency [WJ95]. Note that it is obvious that mental agency actually inherits experience from the symbolic AI. This constitutes thus a compromise between the main streams of AI (see section 2.1).

Of the best known mental agents are the BDI agents, based on the BDI theory (*belief*, *desire* and *intention*) of Rao and Georgeff [GR95].

Reactive Agents. Reactive agents are based on the idea that *the World is the best model* [Bro94]. Intended to handle basic behaviors, they base their architecture on simple routines schemes, excluding abstract reasoning. These simple actions seem indeed easier to perform basic behaviors. The methodology is bottom-up: agents react to changes in the environment in a stimulus-response fashion by executing the simple routines corresponding to a specific sensor stimulation.

A famous reactive agent architecture is Brooks' subsumption architecture [Bro86]. Brooks' robot possesses a layered architecture, where each layer represents a simple behavior reacting to specific stimuli. The high layers can control the most primitive layers by filtering their input data and preventing their output data from influencing the effectors.

Hybrid Agents. Some work tries to unify mental and reactive agencies in order to surmount their respective weaknesses. The idea is the following: basically, these agents are steered by their simple routines reacting to basic stimuli. However, the deliberative module controls the reactive one when it wants to perform stimulus-free actions (like reasoning) or to change long-term goals.

Important examples are Müller's InterRAP architecture [Mue96] and Ferguson's Touring Machines [Fer95].

2.3 Characteristics of MASs

As indicated by its name, a *Multi-Agent System*[1] (MAS) is a macro-system comprising multiple agents, each of which is considered as a micro-system. MASs result therefore from the organization of multiple agents within an environment. This organization is done with a specific hypothesis and goal. Actually MAS research expects that an ensemble resulting from the interactions of individual agents possesses a value-added towards each simple agent capability [NN99]. This is the basic motivation. But what are the characteristics of MASs? The following can be cited [JSW98]:

- In a MAS, every agent has a subjective view; it can only have incomplete information of the system, because its viewpoint is limited.
- As a consequence of the first characteristics, no global control is applied. Each agent has its proper state that is not accessible by other participants in the system. This state changes individually as a result of the behavior rules of the agent. These rules are themselves influenced by data sensed in the environment. On its turn, an agent influences the environment by acting on it.
- The data is fully decentralized, distributed in the participating agents and the environment.

At a modeling level, agents in a MAS are concurrent from one another: each agent is conceived to execute independently from the other agents. In many cases, however, this theoretical concurrency is not guaranteed by an underlying implementation that realizes a serial simulation of a system (see for instance [LBDL99], [Mic99]). Still at a modeling level, a MAS integrates a specific organization of its environment in which agents evolve. This organization can be imposed by either the model or the underlying physical layer.

Although sometimes it is possible to conceive a system with a single agent, this would simply reduce a problem to wrapping a program so that it can interact with a user. This has been, for instance, realized by wrapping expert systems for assisting users in making decisions. Hence, MAS applications have several advantages in comparison to single-agent systems:

- The implementation of problems that ask for distributed data and control over them are more naturally realized using MASs.
- They tend to robustness, because the failure of an agent can be overcome if another agent takes in charge the uncompleted work.
- Scalability: a MAS should easily be extended by adding new agents.
- Reusability: thanks to their modularity, agents should be easily re-integrated in a new MAS.

[1] Several reviews on research in multi-agent systems are available; see for instance [Bra97], [Zie97], [HS98], or [Wei98].

If MASs present real benefits, their construction shows several inherent problems that challenge the research. Some of these problems are applicable to MASs of every type; others are related to specific views. Among the basic questions that should be solved are the following: Which communication schemes should be chosen? What are the means of interaction within a MAS? How should communication and computation be balanced? When agents tend to a common goal, how should they coordinate with one another? How is conflict between agents handled?

The list of questions is by far incomplete. However the main problem remains a software engineering one: how can practical MASs be modeled, designed and implemented? This book contributes to an answer to this question.

We do this in the following way:

- We sustain, in the next section, that the modeling of a MAS should explicitly integrate the interaction setup of the MAS by clearly identifying what we call objective and subjective dependencies, and we insist upon the management of objective dependencies, leading to what we denominate objective coordination.
- We present, in section 2.5, a specific class of MASs as our target and show how it can be modeled.
- We discuss, in section 2.6, actual proposals for helping the implementation of MASs, and, as we promote a focus on objective coordination, we advocate in section 2.6.3 the use of coordination models and languages for designing and implementing objective coordination in a MAS.
- We present in the subsequent chapters a concrete coordination model named ECM and a corresponding coordination language named STL++ that we have tailored towards our target MAS.
- Finally, an implementation of an application of our target MASs is presented using STL++.

2.4 Modeling MASs

Interaction between agents is absolutely essential in a MAS, because it is the "raison d'être" of the MAS. If agents are not able to interact with one another, no global behaviour in the MAS is possible. One has therefore to model the interaction setup of the multitude of agents participating in the MAS. If this is not done, this typically leads to an *agent-oriented* view of a MAS [COZ99b]. This agent-oriented view models a MAS by describing the intra-agent aspects, such as the agent's representation of the world, its beliefs, desires, intentions, and by neglecting the description of the agent interactions and of the space where these interactions take place.

This necessity for a clear identification of the interaction setup in a MAS naturally calls for a separation between the design of the individual tasks

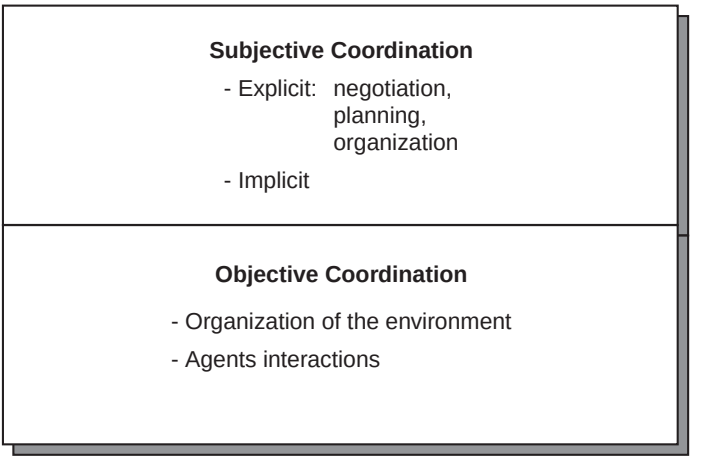

Fig. 2.3. Coordination in a MAS.

of each agent and the design of their interactions. Several researchers have recognized this point, by proposing to explicitly integrate the interaction setup and the handling of the resulting dependencies in the analysis phase of a MAS [Bur96], [Sin98b], [WJK99], [COZ99b], [COZ99a]. We maintain that this can be done at two different levels, according to the types of dependencies.

Indeed the modeling of the setup of multiple agents into a MAS leads to the detection of many dependencies of different nature. On one side, these dependencies rely on the result of the external composition of multiple agents into an ensemble; on the other side, they result from the individual or peculiar point of view of each agent interacting with other agents. We thus distinguish between two types of dependencies [SCH99] and, as coordination is defined as *managing dependencies between activities* [MC94a] (see section 3.1), two corresponding types or levels of coordination (see figure 2.3):

– A MAS is built by *objective dependencies* which refer to inter-agent dependencies, namely the configuration of the system in terms of the basic interaction means, agent generation/destruction and organization of the environment. We refer to the management of these dependencies as *objective coordination*, because these dependencies are external to the agents. Thus, objective coordination is essentially concerned with inter-agent aspects.
– Agents have *subjective dependencies* which refer to intra-agent dependencies towards other agents. The management of these subjective dependencies refers to what we call *subjective coordination*. Thus, subjective coordination is essentially concerned with intra-agent aspects.

Subjective coordination is dependent from objective coordination, because the first is based on and supposes the existence of the second. The

mechanisms that are engaged to ensure subjective coordination must indeed have access to the mechanisms for objective coordination. If this is not the case, no subjective coordination is possible at all. This does not mean that objective coordination belongs to the intra-agent view, but only that the access mechanisms have a subjective expression in the agent. This is essentially the case for mechanisms like sending or receiving information.

We thus maintain that, when modeling a MAS, the first step is to identify which objective dependencies are present and how they can be handled. It is only after this identification that all subjective dependencies can in turn be identified and described. Not differentiating the two levels of coordination leads to MASs that resolve objective coordination with subjective coordination means, i.e. by using intra-agent aspects for describing system configurations. For instance, a MAS intended at modeling the hierarchy in an organization would model this hierarchy internally in each agent by means of knowledge representation of the hierarchy, and would not describe it by establishing the communication flows that represent it.

This mixture of subjective and objective aspects is typically present in MASs composed of mental agents that use agent communication languages (ACL) [LFP99] in order to communicate. The principal aim of ACLs, such as KQML [FFMM94], [LF97] and the FIPA's proposal [FIP99b], [FIP99a], is to offer means for exchanging information and knowledge. But they do this by integrating in a message simultaneously two types of information: (i) lower-level communication information (such as the identities of the participating agents, a unique communication identifier, or the used protocol), and (ii) meta-information about the content of the message, such as the general intention or the type of a message in term of speech acts [Sea69], [Sin98a] (examples are *request*, *deny*, *propose*, *confirm*), and a common understanding, possibly described with ontologies [Gru93] identifying a common semantics between the agents.

In the next sections, we discuss more in detail both objective and subjective coordination. Actually, we classify existing general MAS issues into the two proposed dimensions.

2.4.1 Objective Coordination

Objective coordination is mainly concerned with the organization of the world of a MAS. This is achieved in two ways: i) by describing how the environment is organized, and ii) by handling the agent interactions. We address these two points.

Organization of the Environment. The organization of the environment varies according to the space the MAS wants to integrate. We thus propose to distinguish between *implicit* and *explicit environment organizations*.

An implicit organization does not explicitly model the environment, because it is given or imposed by the underlying logical structure on which a

MAS evolves. This is, for instance, the case for network-aware agents roaming on the World Wide Web, where the environment is structured by the nodes of the network.

An explicit organization, however, establishes a model of an environment that does not necessarily reflect the intended logical structure. This can be done by realizing an approximation of the target. When, for instance, one wants to simulate a continuous physical space, he will not be able to keep the continuity and will have to render the space discrete.

But, more importantly, the environment allows the arising of the interactions between the agents. We discuss hereafter how this can be completed.

Agent Interactions. Handling agent interactions asks for the description of the interactions between an agent and its environment, and the interactions between the agent themselves.

As the environment also have a container function, it can be used to communicate. Indeed, an agent has a relation with its environment by means of its perception. Furthermore, it can influence the state of its milieu with specific actions. The interaction with the environment can then be understood and used to communicate: all information is transmitted within the environment. Communication thus becomes an action that influences the environment. Consider, for instance, the case of an insect-like robot dispersing in the environment information similar to pheromone. This information can be sensed by another robot that notices it as a trace of the roaming of the first agent.

But the interactions between agents are usually based on specific communication means. From the perspective of the direct concerned recipient of the communicated data, we classify these communication means between agents in four basic paradigms (figure 2.4 pictures them):

- *Peer-to-peer* communication: messages are sent directly to specific agents. This is usually done by identifying the partners, for instance with an email-like address (message-passing-like communication). It is also possible that an intermediate channel takes in charge the transmission of the data, and that the partners of communication do not know from each other.
- *Broadcast* communication: a message is sent to everybody in the MAS. Interested agents can evaluate the received data or ignore it.
- *Group* communication: A message is sent to a specific group of agents.
- *Generative* communication: communication is realized through a blackboard [HR88]: agents generate persistent objects (messages) on the blackboard, which are read by other agents. The reading can be done independently of the time of the message generation; thus the communication is fully uncoupled.

Furthermore, it is possible to distinguish *identified* and *anonymous* communication. Identified communication requires an identity of its partner. In

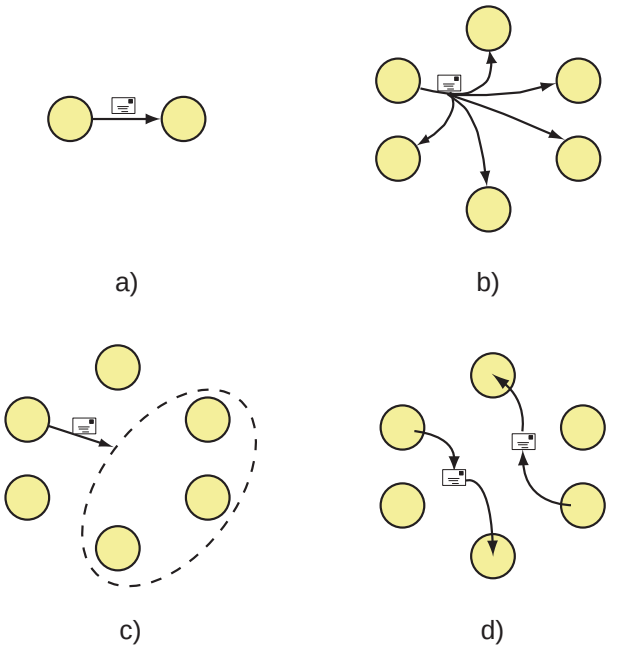

Fig. 2.4. Basic communication paradigms: a) peer-to-peer, b) broadcast, c) multicast and d) generative communication.

anonymous communication, the agent producing the message ignores its recipient, and vice-versa. This can be achieved, for instance, by creating in the agent ports having the role of communication interfaces; these ports, on which an agent puts and reads messages, are connected to other ports belonging to other agents.

2.4.2 Subjective Coordination

We distinguish two types of subjective coordination: *explicit* and *implicit* subjective coordination. Their differentiate themselves in the explicit or implicit treatment of the management of subjective dependencies.

The research in distributed artificial intelligence (DAI) has proposed several coordination techniques [Jen96], [NLJ97], [Oss99] that deal with explicit subjective coordination. These techniques typically consider coordination as *the process by which an agent reasons about its local action and the (anticipated) actions of others to try and ensure the community acts in a coherent manner* [Jen96]. Thus, these techniques are qualified as subjective coordination, because they try to resolve subjective dependencies by means of intra-agent structures that often involve high-level mentalistic notions and appropriate protocols. We characterize these techniques as explicit, because they explicitly handle coordination.

In [Oss99], Ossowski classifies them in three families: *multi-agent planning*, *negotiation*, and *organization*. We briefly explain these families:

Multi-agent Planning
: With multi-agent planning techniques, agents build a plan and commit to behave in accordance with it. This plan describes all actions that are needed to achieve the respective goals of the agents. Note also that planning can be centralized or decentralized.
: An overview on multi-agent planning techniques can be found in [Dur96].

Negotiation
: Negotiation has constituted the most significant part of DAI research in coordination. It can be defined as *the communication process of a group of agents in order to reach a mutually accepted agreement on some matter* [BM92]. Often starting from contradictory demands, the negotiation process generates agreements that can be later re-negotiated.
: A review on negotiation techniques can be found in [NLJ97].

Organization
: Organization techniques support a priori organization patterns by defining *roles* for each agent. A role determines the expectations about the agent's individual behaviour by describing the agent's responsibilities, capabilities and authority inside the MAS. An agent then consults its organizational knowledge of its role in the MAS and acts in accordance with it. Although the purpose of organization belongs to objective coordination, the notion of role typically refers to a subjective dependency towards other roles in the MAS. This is the reason why we consider these techniques as belonging to subjective coordination.
: It is worthy to note that these methods generally suffer from lack of dynamicity in the structure of the organization, because the roles are thought as long-term relationships.
: A valuable review on organization theory applied to DAI can be found in [CG98].

Agents may also coordinate themselves implicitly, without having explicit mechanisms of coordination. This may be, for instance, the case in the framework of collective robotics [BHD94], [MM95], [Mat95], in which robots act on the base of the result of the work of other robots. Thus, this result, which is locally perceived through the sensors, allows to resolve a subjective dependency, namely the necessity to sense a specific information in order to act. This kind of coordination, which is also named *stigmergic* coordination, literally means an *incitement to work by the product of the work* [BHD94]. Our target class of MASs typically allows such a coordination (see section 2.5).

2.4.3 Emergence

We tackle now an aspect that does not belong to the modeling of a MAS, but that constitutes its goal: emergence.

Indeed, MASs are interested in an interaction-centered perspective. Dealing with agent interactions leads naturally to the concept of *emergence* of behavior and functionality [Ste94], [ECJS94]. According to Forrest [For90], emergence can be defined as follows: a system's behavior is considered emergent if it can only be specified using descriptive categories which are not to be used to describe the behavior of the constituent components. Two levels of behaviors are thus differentiated: i) the behavior of the constituents, ii) and the global behavior risen from the conjunction of the constituents. The definition shows that it is not possible to specify emergence, because it is only the result of the interactions of multiple agents.

Emergence offers a bridge between the necessity of complex and adaptive behavior at a macro level and the mechanisms of multiple competences at a micro level [For90]. Although emergence keeps a subjective aspect by directly depending on the cognitive properties of an observer [CLC99a], it has the advantage of basing on simple components and at achieving a global behavior by letting agents interact.

2.5 Our Target Class of MASs

The preceding sections have discussed the modeling of MASs. Before discussing implementation issues in section 2.6, we describe a specific class of MASs; this class is attempted to be implemented on distributed architectures. In chapter 8, we will present an implementation of an application of this class of MASs using our coordination language STL++.

The work presented in this book has been developed in the framework of research aimed at solutions for applications in the fields of robotics and distributed systems. For this reason, we are mainly interested in simulated embodied autonomous agents [CKS+98], [CDSH98]. More precisely, we design multi-agent simulations of collective robotics [BHD94], [MM95], [Mat95], [CH99], whereby global tasks are achieved thanks to emergence. Our simulated robots are operationally autonomous agents: there is no centralized control of the behaviour of the agents, because each agent acts by itself on the basis of its perception and in accordance with its rules (sensing and acting are strictly local to the agent). Furthermore no explicit subjective coordination is engaged, because we follow a stigmergic subjective coordination [BHD94] (see section 2.4.2): an implicit coordination is achieved on the basis of the result of the work of each agent. Thus the only interactions are realized through the influences in the environment.

We organize this section as follows. In 2.5.1, we present a generic model [CKS+98] for our intended autonomous agents' system. Then in 2.5.2, we

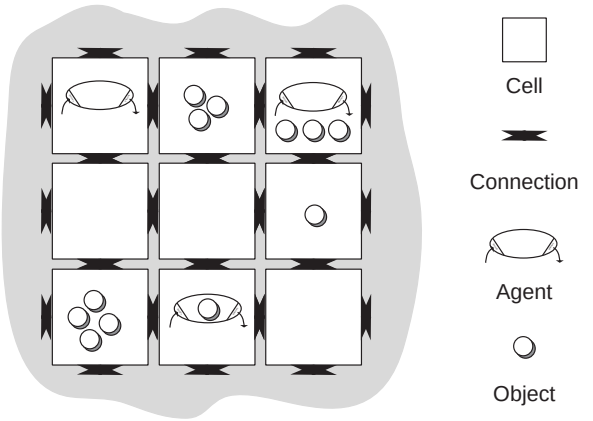

Fig. 2.5. Generic MAS Model.

present a peculiar application belonging to our target class of MASs. This model and its application, which have been defined in the PAI group[2], have been the subject of several studies on the emergence of behaviour [CDS96], [CDSH96], [DCH97], [CLC99b], [CLC99a].

2.5.1 A Generic Model for an Autonomous Agents' System

We first tackle the organization of our generic model by actually describing the objective coordination. The model is composed of an *Environment* and a set of *Agents*. The *Environment* is composed of a set of *Cells*. Each cell includes i) a set of on-cell available *Objects* at a given time, which can be manipulated by agents, and ii) a list of connections to other cells. This list of connections defines a *Neighborhood* which implicitly determines the topology. As this neighborhood can be defined separately for each cell, any type of topology may be achieved. Figure 2.5 sketches the model presented for a four connectivity. An agent possesses some sensors to perceive the environment within which it moves, and some effectors to act in this environment (embodiment). The perception is local, because, at a given time, an agent perceives only on one cell or on a restricted collection of cells. Therefore, interactions between agents are limited to their influences on the environment.

Concerning the intra-agent aspects of the model, a general agent architecture is defined, complying the autonomous agent definition we have adopted in section 2.2. Figure 2.6 sketches the proposed architecture. The modules *Perception*, *State*, *Actions* and *Control Algorithm* depend on each application; therefore, they are under the user's responsibility. The degree of autonomy of an agent is dependent upon the control algorithm module, because this

[2] Parallelism and Artificial Intelligence Group of the Computer Science department at the University of Fribourg.

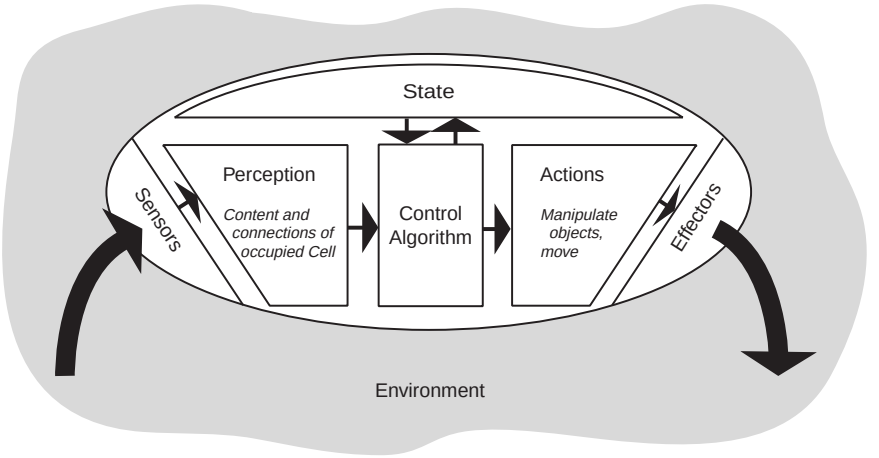

Fig. 2.6. Our target agent architecture.

module determines if an agent is *operationally* or *behaviorally* autonomous [Zie97] (see section 2.2.2). For instance, a very basic operational autonomy would consist of randomly determining which actions to perform; a more sophisticated behavioral autonomy would integrate learning capabilities using, for instance, a neural network.

2.5.2 A Typical Application: *Gathering Agents*

The model presented above can support numerous applications. Among others, it can be used for a simulation of mobile collective robotics [BHD94], [MM95], [Mat95], [CH99]. In fact, we present such a simulation, in which agents simulate the behavior of a real robot seeking objects spread in their environment. As a result of the individual behaviors of each agent, a global behavior is achieved, namely that of having agents stack all the objects, as displayed in figure 2.7.

In the simulation, the agents are operationally autonomous, because each agent in the system acts freely on a cell: the agents choose an action depending on their control algorithm and local perception. The agents do not explicitly negotiate, nor explicitly identify and handle antagonism situations (conflicts between different agents' goals). Every interaction is done in an indirect way through influences upon the environment. Thus, there is no explicit subjective coordination.

The environment is made of a discrete two dimensional L cell-sided grid, a set of No objects and Na agents roaming from one cell to the other. An object is either situated on a cell, or carried by an agent. Several control algorithms are possible (see for instance [CDSH96] and [DCH97] for various proposals).

Fig. 2.7. Collective robotics application: stacking objects.

A simple control algorithm can be explained as follows. Agents roam randomly from one cell to a connected one. As long as an agent does not sense any object in its cell, it moves to another neighbor cell. When it arrives at a cell with *no* objects, two cases are possible: i) if the agent does not carry any object, it will take one object with a probability given by $1/\mathtt{no}^{\alpha}$, where $\alpha \geq 0$ is a constant; ii) if the agent is carrying an object, it will always put down the object.

The simulation presented has been serially implemented [CDS96], [CDSH96], [DCH97], [CLC99a], [CLC99b], showing the emergence of global properties in the whole system: the agents cooperate to achieve a task without being aware of it. In chapter 8, we will present a distributed implementation of the simulation using our coordination language STL++.

2.6 Implementing MAS Applications

Once the model of a MAS realized, one has to lead this model to a concrete implementation. Actually, a very important issue in agent technology has been the elaboration of *languages* for facilitating the implementation of MASs, although a lot of realizations have used ad hoc solutions, without trying to present an abstracted layer on which to build. These languages should fill the gap, often encountered, between the model of a MAS and its implementation.

In this section, we review several important proposals of languages for MASs. We then discuss design methodologies that are emerging from the experience acquired with these languages. Finally we show why these approaches are inadequate and conclude by advocating the use of coordination models and languages for designing and implementing MASs.

2.6.1 Languages for MAS Applications

As most MAS implementations have been built in an ad hoc manner, several authors have proposed *agent languages* that offer a layer of abstraction on

which to develop MAS applications. We distinguish two main trends. On one side, many proposals follow the approach of the agent-oriented programming paradigm [Sho93]. On the other side, concurrent object-oriented programming languages [AWY93], [BGL98] are presented as a base for implementing MASs [GB92], [KB98], [Bri98], [GB99].

Agent-Oriented Languages. Shoham has proposed an agent-oriented programming (AOP) paradigm [Sho93] for programming MASs. This view suggests that agents be directly implemented with mentalistic notions.

In an AOP system, a *formal language* describes the agent mental states. It uses modalities that are more basic than those of BDI[3] [GR95]; they encompass *beliefs*, *decisions* and *capabilities*. The agent's actions are determined by its *decisions* (possibly in the past). On their turn, the decisions depend upon the agent's *beliefs*, which refer to the state of the environment (including other agents' mental states) and the proper mental state, as well as the *capabilities* of itself and others. Furthermore, Shoham slightly modifies the decision notion by replacing it by *commitments*, treating decisions as obligations.

The second element of an AOP system is an *interpreted programming language* named AGENT-0 that is based on the formal language and that concretely programs the agents. AGENT-0 defines an agent in term of its capabilities (what it can do), its initial beliefs and commitments, and a set of *commitment rules*, which describe the agent actions. A commitment is composed of a *mental condition*, a *message condition*, and an *action*. A mental condition is a combination of modal statements referring to the mental state of the agent. A message condition is a combination of *message patterns*. A message pattern is defined by the identification of its sender, the type of message, and a content as a fact or action statement. In order to trigger the rule, the conditions must be fulfilled: the mental condition is matched against the beliefs of the agent, and the message condition is matched against the received messages. An action can be *private* by executing a subroutine or *communicative* by sending a message.

Shoham proposes a last AOP component: an *agentifier* allowing the translation of neutral devices into programmable agents. It is a translation process that applies the reverse method of situated automata [Ros85]. Starting from a low-level description of the device (using a process language), the agentifier has the task of producing an intentional description of the device; it is therefore an "agentification" process. This should permit the integration of existing devices in an AOP system.

A disadvantage of AGENT-0 is that the correspondence between the formal language and the interpreted programming language is loosely defined. CONCURRENT METATEM [Fis94], [Fis97] enhances this drawback by providing an agent language that faithfully executes its associated formal language. In fact, the behaviour of an agent is defined with a set of temporal logics specifications [Eme90] represented by logical rules. And this specification is

[3] Belief, desire, and intention.

straightly executed in order to produce the behaviour of the agent. As the execution of an agent is described by temporal logics which models time as a sequence of discrete states, this execution depends at each moment on the history of the agent.

AGENT-0 and its agent-oriented programming paradigm have inspired and influenced a lot of other work. Thomas has developed PLACA [Tho93] which improves AGENT-0 with planning and communication requests for action via high-level goals. AGENTSPEAK [WRR94] allows agent programs to be written and interpreted in a manner similar to that of horn-clause logic programs, integrating aspects of concurrent-object technology.

Concurrent Object Oriented Languages. A lot of applications implementing MASs use an existing programming language as a basis for their realization and do not employ a language tailored toward their implementation. In a discussion about the relationships between agent languages and other programming paradigms [Mey98], Clark promotes the idea of basing agent programming languages on Concurrent Object Oriented Languages (COOLs) [AWY93], [BGL98]. This point of view is also argued, for instance, in [GB92], [KB98], [Bri98] and [GB99], where the authors are convinced that COOLs are especially useful as a basis for implementing MASs, although these languages are not directly agent languages. This approach is precisely taken by APRIL [MC94b], as we shall see later in this section.

COOLs combine concurrency with object-orientation, leading to the notion of *active object*. An active object adds to the object design the capacity to have its own activity. This leads to programs with inter-object concurrency. Certain languages also allow degrees of intra-object concurrency, i.e. several threads of execution inside the same object[4].

While the passive object model follows a reactivity principle (methods of an object are invoked synchronously by reception of a message), active objects are active since their creation. Some COOLs keep reactivity, like ACT++ [KL90]. But most of them respect more autonomy, by requiring the active objects to accept incoming messages and handling them explicitly. COOLs contain, however, an important drawback to their basic aim of being object-oriented, particularly in what concerns code reuse by inheritance. This drawback is called the *inheritance anomaly* [MY93], [McH94]. Essentially it is originated in the interference between inheritance and synchronization constraints, which often causes re-implementation of synchronized code.

The property of encapsulation of a private state and the support for a proper thread of execution make active objects an attractive layer on which to build autonomous agents. This has led to the use of several COOLs to implement agent-based applications. The COOLs that implement the ACTOR model [Hew77], [Agh86] have a strong influence in this respect [KB98]. As showed in figure 2.8, an actor communicates asynchronously by sending messages to other actors whose addresses it knows. At the receiver actor, the

[4] For a discussion on the levels of object concurrency, see [Weg90].

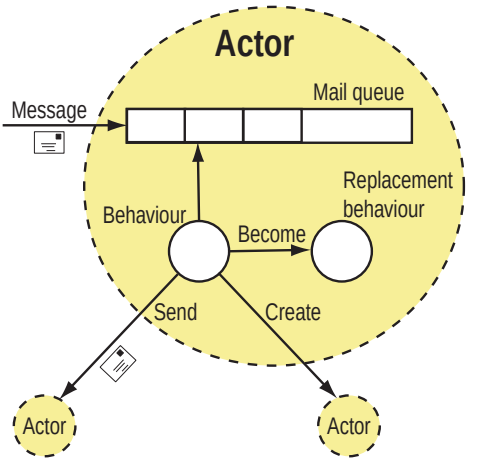

Fig. 2.8. The ACTOR model.

reaction to an incoming message is determined by its behaviour defined by a script. This behaviour allows an actor to send new messages, create new actors, and determine its own subsequent behaviour by the next message arrival. The ACTOR model supports both inter-actor and intra-actor concurrency. Several implementations of the ACTOR model exist, see for instance [Bri89] and [KML93].

A well-known example of object-oriented concurrency for MASs is APRIL [MC94b]. This language is a strongly typed object-based concurrent language with active objects as processes. It is oriented towards the implementation of multi-agent applications. Thus, it is not, as such, strictly an agent language, but it can be applied to implement further agent-oriented languages. Actually, APRIL has been used to realize MAIL [HSM94], a high level language that can be used for implementing MASs. Fundamentally, APRIL defines means for process creation and inter-process communication in a message-passing style, possesses higher-order features that allow program structuring with modules (records of functions and procedures), and permits the sending of code in messages.

We have identified two main groups of existing languages for implementing MASs. On one hand, agent-oriented languages such as AGENT-0 have a typical intra-oriented view of MASs, because they implement a MAS by specifying the mental structures of the agents. They do not support an emphasis of objective coordination in the development of a MAS. On the other hand, COOLs are only considered as a basis for implementing MASs, because the active object model seems adequate for supporting the implementation of agents. But originally COOLs do not directly tackle the space organization of the active objects and their coordination. For that reason, some

enhancements around the basic ACTOR model have been proposed, for instance by structuring the actors in *Actor Spaces* [CA94], which are passive containers, or by organizing them in *Casts* [VA99], which are groups handled by a director. Another approach uses *Synchronizers* [FA95], which allow declarative constructs of coordination activity between several actors. These enhancement enter the framework of the research on *coordination models and languages* [CG92a], [PA98c], which precisely give necessary abstractions for handling objective coordination. But before discussing coordination models and languages in the next chapter, we review methodologies for the implementation of MASs and discuss the appropriateness of coordination models and language for MASs.

2.6.2 Methodologies for MAS Applications

The development of numerous agent languages had the advantage of bringing the theories of agenthood on a practical ground by offering concrete means to program MASs. This has supported the beginning of an engineering view of multi-agent system construction. Nevertheless, the need for methodologies has recently arisen as a help for the design of MAS applications [FMS+97], [IGG98].

Several methodologies have been, to date, proposed, all remaining at an experimental stage [Oss99]. They concern mostly the design of mental agents systems. Furthermore, none has yet gained a large acceptance. [IGG98] reviews several approaches to agent-oriented methodologies, all mostly extending other existing methodologies. Two main groups have been proposed, which are respectively based on knowledge engineering or object-oriented methodologies.

Research extending knowledge engineering methodologies is intended for mental agents. They center their design in the *knowledge acquisition process*. They present, however, an important drawback, by conceiving a knowledge based system as centralized. Two important works are the COMOMAS [Gla96] and MAS-COMMONKADS methodologies [IGGV97].

Object-oriented languages are often used as a natural framework for the construction of agent-based applications. Furthermore, the relation of agents to active objects in the COOLs has already been discussed in section 2.6.1. Therefore, it was natural that methodologies such as the *Object Modeling Technique* (OMT) [RBP+91] have been proposed as a base for agent-oriented methodologies. In [Bur96], for instance, Burmeister proposes a methodology based on OMT. The author defines three models and appropriate process steps for the design of multi-agent applications, all conceived in a mental perspective. First, in the *agent model*, agents are described by their beliefs, motivations, plans and cooperative attributes. Second, the *organization model* uses an OMT notation for stating the inheritance hierarchy and the roles of each agent. Third, a *cooperation model* identifies partners, messages and used

protocols, all considered in a mental dimension. Another proposal based on object-orientation is, for instance, the method for BDI agents [KGR96].

The presented methodologies concentrate therefore on intra-agent aspects and do not really tackle objective coordination as a design dimension. In the next section, we motivate the use of coordination models and language for designing and implementing MASs.

2.6.3 Using Coordination Models and Languages for Designing and Implementing MASs

The research has lead to solutions for facilitating the design and the implementation of MASs. These proposals try to fill the gap, often encountered, between agent theory and agent applications. However, as noted in [COZ99a] and [COZ99b], the proposed solutions are mostly concentrated on intra-agent aspects. They generally design MASs in terms of the internal agent structures, based on mentalistic notions such as the BDI model [GR95], and concentrate on the resolution of subjective dependencies using several (often useful) coordination techniques. Generally, however, they do not explicitly design and treat objective coordination in the MAS, or they do it implicitly by considering, most of the time, only peer-to-peer communication schemes.

However, we have seen that, when modeling a MAS, several objective dependencies clearly appear. The modeling and the resulting design of a MAS is therefore facilitated by separating what concerns inter-agent aspects (objective coordination, i.e. the description of the environment and the agent interactions) from intra-agent aspects (agent tasks and subjective coordination). This idea analogously follows the proposal of Carriero and Gelernter [CG92a] which advocates that programming can be understood as the combination of computation and coordination (see section 3.2.1). Actually, objective coordination is orthogonal not only to the agents tasks, but also to subjective coordination, because this latter handles intra-agent aspects that are in relation with other agents.

Thus, we sustain that it is only on the basis of this separation that one can appropriately design and implement a MAS. And, like a few recent work such as [COZ99a] and [COZ99b], we maintain that this design and implementation can be done with an appropriate coordination model and language.

Indeed, research from concurrent and distributed computing, programming languages and software engineering has developed several *coordination models and languages* [CG92a], [PA98c] (discussed thoroughly in the next chapter). They are suitable precisely for supporting the design and the implementation of objective coordination in MASs, because they define abstractions dealing with the management of the interactions between activities and the organization of the activity space. We therefore propose the use of a coordination model to sustain the design phase of a MAS, and the use of a

corresponding coordination language (the linguistic embodiment of the coordination model) to implement the MAS.

The use of a coordination model for designing a MAS is also promoted in [COZ99a] and [COZ99b], but the authors go even further by following a stronger societal view of MASs [ST95], [Sin98a], [HOY+99]. They maintain that a MAS design should define social laws that regulate the interactions between the agents. These laws should be extensible by having the possibility to specify new rules regulating the communication means. This may be achieved by defining in the interaction space new rules reacting to and ruling communication events. Actually, the authors propose a coordination model that integrates a so-called *programmable coordination medium* [DNO97] (see section 3.5) enabling the definition of new rules detecting and reacting to communication events.

Employing a coordination model and language becomes even more crucial if a MAS runs on a distributed and concurrent architecture, which should be the natural substrate for its execution. In fact, coordination models and languages are utilized for implementing applications executing on such architectures, and have therefore a strong expertise in concurrent and distributed computing. We explain this point.

At a conceptual level, agents in a MAS run concurrently and are organized in several activity spaces. It seems thus obvious that concurrent and distributed architectures should constitute the perfect substrate for MASs. However, the projection of a MAS onto such architectures is not trivial [DC97]. This is especially true for the class of MASs we are interested in, namely simulated-embodied autonomous agent systems, because they possess an explicit organization of an environment (see section 2.4.1) that simulates a continuous space, and this organization does not comply with the structure of our underlying architecture. This may be the main reason why most implementations of this class of MASs are based on sequential architectures with round-robin mechanisms. This is, for instance, the case of important toolkits such as SWARM [LBDL99], SIM_AGENT [SP95], [Slo99], and the KHEPERA Simulator [Mic99].

At an implementation stage, we thus have to deal with the organization of the actual processes, the active pieces of software which represent the agents, along with the inherent problems of shared resources and data, synchronization and consistency concerns. Coordination models and languages are precisely aimed at providing solutions for these problems: they use coordination media and tools so as to deal with the consequences of the concurrency and the spatial distribution of the supporting processes.

Therefore, to realize a MAS on a concurrent and distributed architecture in the most natural way, an appropriate coordination model and corresponding language are needed. On one hand, the coordination model should reflect the image of the MAS model (objective coordination at the modeling level) with the utmost fidelity, while preserving the structure of the agents

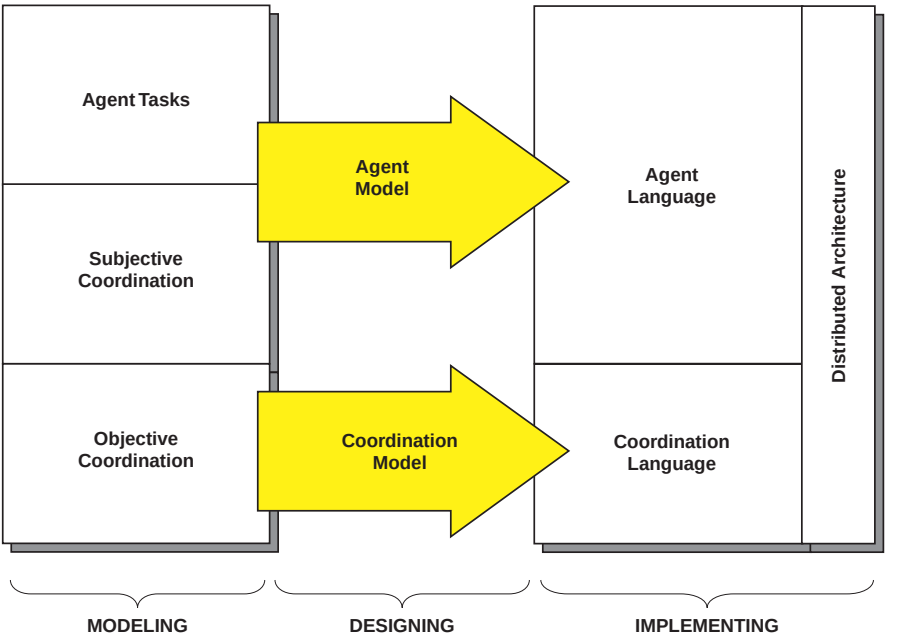

Fig. 2.9. Implementing MASs.

and their behavior, such as, for instance, their degree of autonomy. On the other hand, the coordination language must enable a mapping of the MAS on a distributed and concurrent architecture by managing the creation and destruction of the processes that support the agents and by insuring their interactions and organization within structured and hierarchical spaces.

In conclusion, our approach, which is sketched in figure 2.9, can be summarized in three steps:

Modeling: One first analyzes which objective coordination is needed in the MAS. It is only on this basis that one should identify two remaining crucial issues, namely the agent tasks, as well as the arisen subjective dependencies and, if necessary, the handling of these dependencies.

Designing: An appropriate coordination model is applied for designing the objective coordination in the MAS, and an agent model is also applied for designing agent tasks and subjective coordination.

Implementing: The realized design is implemented using a corresponding coordination language and, respectively, an agent language, which can be, in some cases, a common computation language.

In the next chapter, we thoroughly review what are coordination models and languages. Subsequently, we present the ECM coordination model and its instance STL++.

3. Coordination Models and Languages

3.1 What Is Coordination?

Coordination is a central problem for numerous complex dynamic systems composed of interacting activities. In order to study issues related to coordination, Malone and Crowston proposed the development of a new theory called *Coordination Theory* [MC94a]. This theory intends to study how coordination is handled in various systems such as in enterprises, animal colonies or markets [Kle97]. It is therefore interdisciplinary, re-grouping ideas from sciences such as computer science, organization theory, economics, linguistics or biology. By providing practical cases from these different sciences, the authors try to identify common key concepts of coordination in order to facilitate a cross-fertilization of these different disciplines.

Diverse definitions of coordination have been proposed. Probably the most general has been stated by Malone and Crowston [MC94a]:

Coordination is managing dependencies between activities.

This definition means that all participants in a coordination process have interdependencies. Thus the elaboration of coordination theory has tried to identify which generic dependencies can exist and which sort of processes may be involved.

An interesting general outlook on coordination can be found in [Oss99], in which Ossowski gives an overview on formal and informal characterizations and models of coordination in human societies and its models in social science.

3.2 What Are Coordination Models and Languages?

3.2.1 Motivation

The fields of concurrent and distributed programming, programming languages and software engineering have also recognized the need of handling coordination. In particular, the research has focused on the definition of several coordination models and corresponding coordination languages [CG92a], [PA98c]. Based on a model, a coordination language provides means to compose and control software architectures consisting of concurrent components

M. Schumacher: Objective Coordination in MAS Engineering, LNAI 2039, pp. 33–49, 2001.

[ACH98]. Its basic goal is therefore to handle the interactions among concurrent activities.

The starting consideration of this research has been stated by Carriero and Gelernter by advocating the following slogan [CG92a]:

programming = computation + coordination

This means that a clear distinction between computation and coordination is a key element in the design of concurrent programs. The lack of separation between the computation elements and their interaction makes a program difficult to be implemented and understood. This separation has therefore the advantages of facilitating the reuse of the components and of trying to identify patterns of coordination that could be applied in similar situations (see for instance [FK97]).

These considerations lead to a specific definition of coordination in the context of this research [CG92b]:

> *Coordination is the process of building programs by gluing together active pieces.*

This definition shows that coordination models and languages typically belong to *middleware*. However, other middleware solutions for implementing concurrent and distributed applications mostly concentrate on low-level communication mechanisms. Most known platforms include message passing libraries such as PVM [Sun90], [GBJ+94] and MPI [GLS94]. The distributed objects platform CORBA [OMG97b] is also very used. Coordination languages are viewed as the linguistic counterpart of these approaches.

3.2.2 Key Elements

A *coordination model* is an abstract framework for the composition and interaction of active entities. It proposes solutions for dynamic process creation and destruction, communication mechanisms, multiple communication flows and activity spaces.

A general coordination model can be defined by the following elements [PA98c], [ACH98], [Cia99]:

- The *coordinable entities*, which are the active entities running concurrently. They are the direct subject of coordination, therefore the building blocks of a coordination architecture.
- A *coordinating medium*, which allows the coordination of all participating entities. This medium also serves to assemble entities into a configuration. It is therefore the actual space where coordination takes place.
- The *coordination laws*, which specify the semantics framework of the model. They determine how the entities are coordinated using the coordinating media.

A *coordination language* is the materialization or the *linguistic embodiment of a coordination model* [CG92a]. It offers syntactical means with which a coordination model can be used for implementing an application. The coordination language is orthogonal to a computation language in the sense that it is only concerned with coordination purposes. Hence coordination languages are not general purpose programming languages.

The concept of separation of concerns requires principally that the designer of a concurrent application identifies the two parts and uses appropriate means for implementing coordination. Although originally, Carreiro and Gelernter introduced the term 'coordination language' to designate systems supporting compilers, this expression has evolved to designate also simple language extensions realizing a specific coordination model, for instance using macros (applying therefore a pre-compiler) or a library. In this second case, the coordination code is usually intermixed with the computation code, but mixture is only stylistical or at language level.

There exist several dimensions that can be used to classify the numerous coordination models and languages. For instance, they can be classified in *endogenous* and *exogenous* models [Arb98], whereby the criterion of the categorization is the integration or not of the coordination mechanisms in the computation. But the most used classification distinguishes between *data-driven* and *process-oriented* coordination models [PA98c].

Based on this latter taxonomy, the next sections review the two families of models, basing on the cited paper. We do not try to give an overall review of these models, but rather describe the most characteristic of each family and concisely discuss several other proposals. For a deeper understanding, an interested reader can refer to the cited papers.

3.3 Data-Driven Coordination Models

In data-driven coordination models, the state of a program is *defined in terms of both the values of the data being received or sent and the actual configuration of the coordinated components* [PA98c]. This means that processes are in charge of handling the data and coordinating themselves with the other processes.

This double function of the processes introduces a stylistical mixture between the coordination and the computation code. Although a language of this class of models offers clear means for the communication and the configuration of the components, a certain difficulty to distinguish between coordination and computation may consequently appear. These models delegate to the user the task of organizing the code so that the coordination remains clearly identified.

Data-driven coordination models are therefore mostly concerned with *data*. This is the reason why most of them are designed primarily using a

shared dataspace [RC90] that functions as the unique mean of communication: this shared dataspace is the coordination medium through which the coordinables exchange data structures. Data can be put, retrieved or copied from it. This access to the data and its representation is determined by the coordination laws of the coordination model. Communication through a shared dataspace is often referred to as *generative communication*.

It is important to see that the generative communication scheme allows processes to be uncoupled in space and time. Producers and consumers do not have to synchronize in order to exchange data. The existence of a consumer is not even required when data is posted on the dataspace. But a consumer will have to wait until a producer creates some desired data. This means that this style of communication is anonymous, because the communicating processes do not necessarily know the identity of each other.

Proposed data-driven models differentiate themselves in several aspects. However, most of them belong to two major groups: models derived from LINDA [Gel85], [CG89] and models based on *multiset rewriting*. Note that a few models cannot be classified in these two families. They are not treated in this review. An interested reader can refer to [PA98c], which discusses some of these models.

3.3.1 LINDA

The principal characteristics of LINDA [Gel85], [CG89] is its *generative communication* style. To communicate, processes do not send messages or share a variable, but they communicate through a shared data space. The communicated data items are called *tuples*. A tuple is a non-empty sequence of data items of simple data types. Tuples are generated by a producing process on a *tuple space*, which is a multiset of tuples. Later, a consuming process can consume or read a tuple.

The creation of concurrent processes uses the same mechanism. A creator puts on a tuple space an *active tuple*. Active tuples are tuples associated with processes that perform a computation. When an active tuple terminates, it returns a result that becomes an ordinary data tuple (*passive tuple*). Note that the simplicity introduced by the homogeneity between data and process generation is one of the advantages of LINDA.

Tuples are looked up associatively, using a *tuple matching* mechanism. In order to retrieve a tuple from the space, a consumer must use a *template*, which is called an *anti-tuple*. A template can consist of either *actual* or *formal* data items. An actual data item is a constant of a specific type. A formal item is a placeholder for a desired type. The tuple matching functions as follows: a tuple matches a template if both have the same arity, and if each tuple data item matches each template data item at the same position: an actual item matches a tuple item from the same type and value, a formal item matches a tuple item from the same type but with any value.

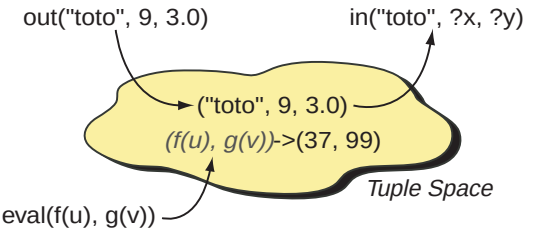

Fig. 3.1. A LINDA tuple space example.

LINDA defines several primitives, which are used for storing and retrieving information from the tuple space, and which must be integrated in a host language. Several robust implementations have been realized, integrating these primitives in conventional programming languages, for instance C-LINDA [CG92a] or Eiffel-LINDA [Jel90]. The LINDA primitives can be re-grouped in three categories, namely communication (`out`, `in` and `rd`), predicates (`inp` and `rdp`) and process control (`eval`) primitives.

`out` allows one to store a tuple in the tuple space. As the space is a multiset (or bag), identical tuples can coexist. `in` retrieves a tuple from the tuple space using an anti-tuple. The formal items are instantiated by the actual values of the retrieved tuple. If several tuples match the anti-tuple, one of them is chosen non-deterministically. The `in` operation is blocking: it returns only when a matching tuple is found. `rd` functions like the `in`, with the difference that it does not retrieve a tuple, but makes a copy of the matching tuple.

`inp` is the predicate version of the `in` primitive. It returns a tuple only in case there is one. If no tuple is present at the moment of the invocation, the primitive returns a failure at once. `rdp` has the same non-blocking semantics as `inp`, except that it reads a tuple.

`eval` is used for the creation of active tuples. In an active tuple, at least one element is a function returning a value. Each function is computed with its proper thread of control. After each function has terminated, the active tuple becomes a passive tuple created with the returned values in the same order as the functions. The tuple only becomes visible at this moment: before completion, the active tuple was not "visible" for other primitives.

Figure 3.1 shows an example of the use of LINDA primitives. A process posts on the tuple space the tuple `("toto", 9, 3.0)` whose items are all actual. Another process retrieves this tuple with `("toto", ?x, ?y)`. The template used contains two formal items `?x` and `?y`, which are bound to `9` and `3.0`. The operation `eval(f(u), g(v))` creates an active tuple. After completion this active tuple becomes a new passive tuple, in our case `(37, 99)`, `37` and `39` being the returned values of respectively `f(u)` and `g(v)`.

LINDA has been applied extensively, for instance with PIRANHA [CFGK95] for supporting adaptive parallelism (parallel computation on a dynamic set of

processors). Furthermore, LINDA has inspired many new coordination models. We discuss some of them in the next section.

3.3.2 LINDA-Based Models

Many coordination models have evolved around the first and most influencing coordination model, namely LINDA. These models are constructed around a shared dataspace whose elements are tuples. The semantics on the manipulation of tuples and even the evolution of this data form differentiate the models. They all have tried to address several drawbacks of the original LINDA.

The most important enhancement, which was soon recognized, is the need of modularity of spaces. This has led to several proposals for supporting multiple spaces [Gel89], [Jen94], such as BONITA [RW97], LAURA [Tol96], POLIS [Cia91] or SONIA [Ban96].

Another improvement of LINDA can be achieved by offering more flexible primitives in order to better the expressivity and the performance of the manipulation of tuples. This is, for instance, achieved by BONITA [RW97], which defines a finer grain notion of tuple retrieval by separating the search for a tuple from the checking of its arrival. This separation allows to launch several parallel requests on tuple spaces without being blocked after each invocation. An id is then used for checking the arrival of the tuple. This scheme allows important gains in performance. Note that BONITA also supports multiple tuple spaces and the manipulation of groups of tuples for moving them from one space to the other.

Due to its generative communication style that decouples processes in time and space, LINDA seems to have strong advantages for supporting the implementation of open systems. But in spite of its uncoupling character, LINDA may present several problems for implementing them.

In open systems, security requirements are of particular importance, because such systems are highly dynamic and not predictable. However, the communication in LINDA is very unsafe, because every process can potentially access every tuple. Several works have tackled security issues of LINDA. [vdGSW97], for instance, suggests the use of strongly typed spaces, field access patterns, tuple space access patterns and assertions. [BOV99] introduces cryptographical methods, proposing the SECURE OBJECT SPACE model, in which a data element can be locked with a key and is only visible for a process possessing a matching key.

But security issues may not be the only difficulties of LINDA when implementing open systems. As, in open systems, services may dynamically appear and disappear, one sometimes want to ensure that a tuple is the result of a specific corresponding `eval`. This is precisely achieved by LAURA [Tol94], [Tol96], whereby processes exchange services through a shared *service space* (analogous to the LINDA tuple space). The elements contained in the service space are *forms* for service offers, requests and results. A process providing a

service puts a service offer form in the space. Another process interested in a service posts a service request form onto the space; this will start a search for a matching service. If both match, the corresponding service is executed and returns a service-result form.

Another important enhancement of LINDA is its integration in an object-oriented paradigm. This is the main aim of OBJECTIVE LINDA [Kie97], in which tuples become *objects* and tuple spaces *object spaces* (OS): the coordination is achieved using the production and consumption of objects over nested object spaces. As objects are handled as encapsulated entities, the matching is based on object types (defined using *Object Interchange Language* (OIL), a language-independent notation) and on predicates defined by the type interface. As in LINDA, OBJECTIVE LINDA distinguishes between *passive* and *active objects*. A passive object is a passive data that can be stored in an object space. When it is in such a space, it is not possible to call methods on it. Active objects execute their own activity operating on object spaces. In order to support dynamic composition of spaces, OBJECTIVE LINDA allows to assign dynamically another OS to an active object. This is achieved by using so-called *object space logicals*, which are references to OSs that can be passed around by active objects. The operations on tuple spaces furthermore include a timeout parameter, which allows a compromise between communication that immediately returns and infinitely blocks: this is very useful for open systems, in which resources may dynamically disappear.

OBJECTIVE LINDA has shown an integration of LINDA in an object-oriented paradigm. Recently, another integration proposal has been made, namely JAVASPACES [Sun98]. Based on a matching mechanism, it allows to put and take groups of objects from an object space. A particularity is that objects are stored persistently in the space and that transactions over multiple spaces are possible. A notification mechanism is also proposed, allowing active objects to ask the space to notify them when a specific object, which should match a given template, is written in the space.

3.3.3 Models Based on Multiset Rewriting

Models based on multiset rewriting [BM93] constitutes another group of data-driven models. The coordination medium is composed of multisets, i.e. sets whose elements can have multiple copies. A coordination model is then concerned with the manipulation of the multisets, mostly defining rewrite rules.

GAMMA (General Abstract Model for Multiset mAnipulation) [BM90], [BM93] is one of the first proposals of this family of models. GAMMA is based on a chemical reaction metaphor. The model defines a unique operator Γ that attempts to transform the whole data by applying reactions on the multiset. A reaction is defined as a pair (R, A), where R denotes a reaction condition that must be fulfilled in order to execute the action A. An action is a rewrite rule $lhs \longrightarrow rhs$ that selects the elements of the multiset that match lhs and replaces them with rhs. As long as a reaction is possible,

the Γ operator applies it on the multiset until no more reaction is applicable (implicit termination condition). If several reaction conditions are valid at the same time, a reaction is chosen non-deterministically. Furthermore R and A are pure functions. Therefore, if a reaction condition is true for disjoint subsets, the corresponding reaction can be applied simultaneously on each subset, leading to a strong parallelism.

Chemical Abstract Machine (or CHAM for short) [BB92] follows the same metaphor as Gamma by describing computation as a form of chemical reaction. But CHAM principally enhances Gamma by allowing molecules to be encapsulated into solutions. This leads to a hierarchical coordination medium. The interesting property of CHAM is its generality: CHAM is a model that can be used to represent non sequential computations based on multiset rewriting schemes. It can thus be applied to map several coordination models (see [CJY95] for an example).

Interaction Abstract Machine (IAM) [ACP93] is a model close to the CHAM. An IAM is composed of several independent subsystems, named agents by the model. The particularity of IAM is that an agent has a state, defined by a multiset of resources. This multiset can evolve under the action transformation rules (called methods) of the subsystem. A method, which has the general form: $A_1, A_2, ..., A_n \longrightarrow B_1, B_2, ..., B_m$, rewrites the multiset as follows: if the state of an agent contains the resources $A_1, A_2, ..., A_n$, then they are replaced by $B_1, B_2, ..., B_m$. If two different methods can be applied on two disjoints subsets of the state, they will be executed concurrently. In the opposite case, a method is chosen non-deterministically. Concerning the communication, it is done with *broadcasting* resources. Note that one of the drawbacks of IAM is that it does not offer means of structuring the environment.

The object-oriented logic language Linear Objects (LO) [BAP92] is based on Interaction Abstract Machine and uses Linear Logic [Gir87]. In Linear Objects, methods are also rewriting rules, like IAM methods. However the manipulating resources are implemented as tuples with variables. Thus, to apply a method, each variable must be instantiated by pattern matching. The corresponding tuples are then replaced with new tuples.

ShaDe [CCR96], [CR98] is an object-oriented coordination model and corresponding language directly inspired from IAM. Methods of ShaDe classes are the coordination rules describing the behavior of the objects. As in IAM, the state of an object is defined by its associated multiset. It changes when the rules are triggered, the state is rewritten, and the object receives a message. If the head of the method, which can contain Linda `in` and `rd`, is satisfied, the body is executed. The body can write items in the object's multiset, write items in other object's multisets, or create a new object. The sending of messages uses a special form of multicast, because the message is received by each object whose name matches the pattern. In this way, point-to-point, multicast and broadcast communications can be expressed.

Furthermore, messages are persistent: at its creation, an object receives all messages that have been sent before and that match its name.

BAUHAUS [CGZ95], [Hup96] is an extension of LINDA to a multiset rewriting scheme. The model simplifies LINDA by unifying the notions of tuples and tuple spaces in nested *multisets* supporting multiple spaces: a BAUHAUS multiset can contain primitive elements or other multisets. The model also uses a matching mechanism for posting and retrieving multisets but this mechanism is based on set inclusion rather than on the order, value and type-sensitive matching scheme of LINDA: templates (anti-tuples) are not necessary anymore. Unlike in LINDA, there is no difference between passive data objects and active data objects (processes): the `out` primitive is used for both types of objects and processes are represented explicitly in multisets by using tokens. Furthermore, the `in` and `rd` primitives do not have side-effects, but are functions returning multisets.

3.4 Process-Oriented Coordination Models

With process-oriented (or control-driven) coordination models, *the state of the computation at any moment in time is defined in terms of only the coordinated patterns that the processes involved in some computation adhere to* [PA98c]. Therefore an application is centered on the processing or flow of control. The attention is concentrated on processes and their organization. The data does not play a role in the coordination.

This orientation mostly brings with itself a stylistical separation between computation and coordination, providing a new coordination language totally distinct from a computation one. The components being coordinated are considered as black boxes that produce and consume data on well-defined interfaces to the external world, usually referred to as ports. This separation results in a more comprehensible separation of concerns and strongly helps reusability.

Connections between ports are mostly of a channel nature, allowing point-to-point interactions. In addition to these communication means, many models broadcast control messages or events in the environments. These last are used to inform other processes of state information. A typical example is an event that is sent by a process in order to indicate that it has terminated.

The variation of the concepts presented results in several different coordination models. Events can be simple information units or contain data of a specific type. They can be transmitted through the connections or broadcast in the environment. The semantics of the connections is also a varying factor. In addition to that, several degrees of dynamicity are allowed: connections can be totally static or evolve during an application, and ports can be identified by other processes or not.

The next section review the IWIM model of coordination, a typical control-driven model that defines a family of models, and MANIFOLD, a language-binding of IWIM. We then discuss other control-driven approaches.

3.4.1 IWIM

IDEALIZED WORKER IDEALIZED MANAGER or IWIM for short [Arb95], [Arb96] is a general model of coordination. As basic abstractions, IWIM defines *processes*, *ports*, *channels* and *events*.

A process is considered as a *black box*. It communicates through ports of connections, which are named openings in the bounding wall of the process. To communicate, a process writes and reads *units* (data) on its ports. It is not aware to whom information is transmitted to: the communication is anonymous. Two classes of processes may exist: *managers* and *workers*. Workers basically perform a (computational) task. Managers create new processes and regulate the connections between workers by dynamically binding ports of different workers into connections, or disconnecting existing connections. In general, no process determines its communication with other processes by itself. Note also that a process can play both the role of a worker and a manager at the same time. This allows the construction of hierarchies of processes, where the leaves are pure workers.

The established connections are of channel nature. A channel is considered to be *unidirectional*, connecting a producer process (*source* side) to a consumer process (*sink* side), and *reliable*, assuring that units are transferred without loss, error, or duplication. IWIM defines five general channel types, differentiating themselves in their way of handling disconnections and the resulting *pending units* (units created by a producer that have not yet been consumed). An *S Channel* is a synchronous channel, in which pending data never exists and producer-consumer pairs are always required for the existence of the channel. The four other types are of an asynchronous nature. A *BB Channel* (B for break) is disconnected if one of the two sides is disconnected. A *KK Channel* (K for keep) does not disconnect from one side if it is disconnected from the other. A *BK Channel* disconnects from its producer if it is disconnected at the consumer side; there is no disconnection at the consumer side if the producer is disconnected. A *KB Channel* has the opposite semantics of a BK channel.

Reading on a port blocks the corresponding port until a unit is available. Writing on a port blocks as long as the port is not connected, and, in the case of an S channel, until a producer has read the data.

Events are used for information exchange. They usually indicate process state information to other processes. An event is therefore a signal broadcast in the environment, causing an *event occurrence*, composed of the identity of the event and of its producer. Every process in an environment may capture an event occurrence and react to it. However an event is caught only by interested processes.

MANIFOLD. MANIFOLD [AHS93] is a coordination language based on IWIM. To every basic IWIM abstraction corresponds a MANIFOLD language construct. The whole communication is asynchronous, because MANIFOLD uses asynchronous channels, and the launching and reacting to events does not synchronize the corresponding ports. Moreover, the separation between computation and coordination is enforced by clearly distinguishing computational from coordination processes. A computational process may be implemented in any language extended with primitives for producing and consuming units on ports, and raise and receive events. A manager process implemented in the MANIFOLD language do not provide means for computation.

MANIFOLD supports BB, KK, KB, and BK channels with infinite capacity. Only KB and BK streams can be re-connected. On producing an item on a port that is connected to several streams, this item is replicated for each connection. A read on a consuming port connected to several streams will choose non-deterministically an item.

The evolution of the communication topology is *event-driven*, based on *state-transitions*. An event is a signal that is not parameterized and does not transmit data. When an event has been triggered, it is propagated in the environment. An interested process receives it automatically in its *event memory* which is a set of events (only a single copy of identical events is possible). A process can then examine and react to event occurrences.

A coordinator (manager) process is called a *manifold*. Its body consists in several states, composed of a label and a body. The label, which is a condition for the transition to the state, is defined on the event memory as a conjunction of patterns that match observed events. The body of a state defines a set of actions executed at the time of a transition to that state. These actions which are executed in a nondeterministic order can create new processes, broadcast events, post events in their own event memory and create or reconnect connections. After execution of the actions and on satisfaction of a new label condition, all streams that have been constructed and/or (re-)connected in that state are *preempted*, i.e. broken at their B-end, and a transition to the new state is realized.

MANIFOLD has been applied to diverse application domains (see for instance [PA98b] or [EL99]). Furthermore, it is promoted for the support of distributed and parallel software engineering [Pap98]. It has also been proposed for the modeling of activities within information systems [PA98a], whereby the modeling framework presented differentiates three steps in the design: firstly, a graphical representation views the interrelationships and behaviours of every process (this can be done using VISIFOLD [BA96]); secondly, each process and its reactions to launched events are described semi-formally; thirdly, the implementation is realized.

3.4.2 Other Approaches

A coordination language very near to MANIFOLD is CONCOORD [Hol96]. CONCOORD separates processes into computation and coordination processes. Computation processes carry out a sequential computation in any language (extended with few communication primitives), communicating anonymously through ports. They are created and killed by the coordination processes, which are implemented in CONCOORD's coordination language: coordination processes are only concerned with concurrency, managing the dynamic evolution of the program structure by creating channels between ports.

Configuration description languages, like DURRA [WDGL96], the PROGRAMMER'S PLAYGROUND [GSM+95], or RAPIDE [SDK+95], deal with the construction of complex software by interconnecting existing components. They are concerned with the separation of the structural description of the components from their concrete behaviour. For these reasons, they can be considered as coordination languages too, mostly belonging to the process-oriented family.

DARWIN [MNK93], [MDK96] is a typical configuration language. Its principal aim is to permit the dynamical structuring of parallel and distributed programs in groups of processes that communicate with message-passing. DARWIN separates the description of the structure or the interconnections between process instances from the actual code that implements the process computation. To achieve this purpose, processes are defined as *context independent types with well defined interfaces*; thus they only possess references to local entities and refer to ports for communication. A process can export (*provide*) its own ports, which are then used to receive data. A process can also have references to remote ports that it has *required*. Furthermore, it is possible in DARWIN to define *components*, which are specified by a set of processes and/or other component types, a set of instances of these types, and bindings between port variables (provided ports) and port references (required ports) of these instances. A component can also have its own port variables and references. But the most powerful mechanism of a component is its ability to be generic. This genericity is achieved by defining a component specifying the structure of interconnected components, without defining the implementation of each instance. An implementation (a component type) can be assigned to the generic component at runtime; its interface must only be compatible, namely by having the identical number of required references and provided ports, declared in the same order.

TOOLBUS [BK96b], [BK96a] proposes a different approach, by providing a communication bus (a TOOLBUS) to which components or tools can be connected and thanks to which they can be coordinated. Tools, which can be totally heterogeneous, can so be combined into a coherent system. They cannot communicate directly: the interactions between them are handled through the TOOLBUS by controlling them with a *script*. Furthermore, an *adapter* acts as a wrapper of the tools with the task of mapping the tool's

internal data format and message protocols into those of the TOOLBUS. The TOOLBUS is then composed by the parallel composition of processes that represent the tools. These processes are described as expressions including communication primitives, and process creation and composition operators. Note that TOOLBUS has been used for several applications. For instance, a framework has been developed for debugging distributed applications [Oli97]; it is built on existing sequential debugger techniques and implementations.

[Buf97] proposes the CONTEXTUAL COORDINATION model, which is an IWIM-like model that focuses on coordination in distributed objects systems. To that aim, the model separates objects from their execution contexts. A *context* is a group of active objects that have on their turn their own context; it is therefore an encapsulation mechanism that delineates executing worlds. In this hierarchy of contexts, objects interact through connections linking *gates* (output ports) with *methods* (input ports).

The coordination language COLA [HAK94] is intended at the implementation of applications in massively parallel environments. In such systems, the cost of maintaining a global coherent computation space would be very important. For that reason, COLA has no global name space: each process has a locally temporary view of the system (locality). This is achieved using for each process two abstractions: i) a *Range of Vision* defines the set of processes a process can communicate with at a specific moment of its life; and ii) a *Point of View* defines a group of processes in a particular Range of Vision, allowing to abstract communication topologies such as rings or hypercubes. In order to communicate, COLA extends the message passing paradigm by adding to a message: (i) a high-level *descriptor* indicating the sender, the set of receivers, and a set of methods used by a receiver to treat the body of the message; and (ii) a high-level *protocol* describing the communication protocol used to deliver the message.

3.5 Hybrid Coordination Models

The last sections have presented important coordination models and languages, following the classification into data-driven and process-oriented approaches. It is worth to note that most data-driven and process-oriented models can respectively be classified in logically non-distributed (communication with shared data spaces) and logically distributed (communication with message passing) models, which is a taxonomy of distributed computing models proposed by [BST89].

The investigation in coordination models and languages is still an important research area that has many open issues [Cia99]: study of semantics issues of coordination models; extension of existing models or proposal of new models and languages adapted for specific application areas; further investigation of the use of coordination technology for the internet (see for instance [GNSP94], [CKTV96]). An important source of inspiration for appro-

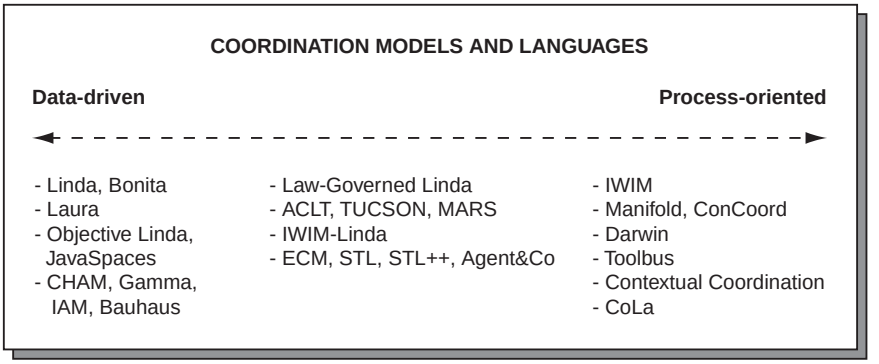

Fig. 3.2. A taxonomy of coordination models and languages.

priate coordination models is the comparison of the models with one another. This may be realized by studying the expressive power of the communication primitives of coordination models (see for instance [BJ99] for a comparison of LINDA-like models), or by comparing models with case studies (see for instance two case studies [dJ97] and [CNT98], and corresponding sample implementations [HPS97], [SvK97] and [Scu99], [RV99]). Both data-driven and process-oriented models present several advantages and drawbacks depending on the application domains. For this reason, the confrontation between data-driven and process-oriented models seems to be especially fruitful. Indeed, several hybrid models [COZ99b] have already been designed by wanting to improve a model of one of the two families or by explicitly confronting and integrating elements from one family in the other. We discuss some hybrid models hereafter. The figure 3.2 summarizes the complete taxonomy presented.

A first group of work integrates the event mechanism into shared dataspace models. This can be done by considering a tuple space as being reactive in the sense that the space can react to communication events rather than to the communication state changes only. Thus, from the point of view of the coordination medium, the observability is shifted from the tuples to the communication operations over the tuples. When an operation is realized on the space, a corresponding reaction is triggered. For most LINDA-derived models, the reactions cannot be modified, because their semantics is fixed by the model. But reactive models allow reactions associated to specific communication events to be programmed, leading to the notion of *programmable coordination medium* [DNO97]. When a communication operation is executed, a reaction catching the event produced atomically executes a sequence of operations which usually have access both to the space and the information associated with the event. The idea of a programmable medium can be found in the $\mathcal{ACLT}$ model of coordination [Omi95], [DNOV96], which extends LINDA

with logic[1], multiple, and reactive tuple spaces. In the context of mobile and network-aware agents, two models also integrate a programmable medium: TuCSoN (*Tuple Centers Spread Over Networks*) [OZ98], [OZ99] and MARS [CLZ98]. Note that, applied to MASs, a programmable coordination media can elegantly support the programming of new laws regulating the communication in a MAS [COZ99a], [COZ99b]. This permits a design focused on the interactions that is even stronger than the view presented in this book, because one can adapt the coordination laws for one's needs.

Law-Governed Linda [ML94] is also a model that follows the notion of programmable media. The basic motivation of Law-Governed Linda is the inherent security problems of Linda that we already have discussed in section 3.3.2. In order to control each exchange of tuples through the tuple space, the model forces every process to adopt specific laws that control each exchange of tuples through the tuple space. For achieving this goal, Law-Governed Linda attaches a *controller* to every process. Each controller, which is in charge of regulating the exchange of tuples, has a copy of the law, and only allows a communication if it conforms to the law. This law regulates the occurrence of so-called *controlled events* that occur at the boundary between a process and the medium. It determines the effect of an event using a prescription, which is the *ruling* of the law, realized with a sequence of primitive operations. An example for the ruling of the law controlling an `out` could be to transform the corresponding tuple by concatenating some useful information.

Another hybrid model that combines control-driven elements with a data-driven model is IWIM-Linda [PA97a], [PA97b]. As indicated by its name, this model integrates the functionality of IWIM in Linda, offering thus secure and private communication channels to Linda. IWIM-Linda is basically composed by three types of processes. Firstly, there are ordinary *computation* processes, which have access to the tuple space by means of Linda operations; they should not have information about the other computation processes. Two classes of computation processes are distinguished: IWIM-compliant processes (i.e. able to communicate via ports, and to broadcast and receive events, all via the tuple space) and non-compliant ones. Secondly, a *monitor* process is charged of intercepting all communication between a process and the tuple space; it transfers sent data to the concerned ports and treats events. Thirdly, *coordination* processes establish the streams between the monitor processes, which represent the computation processes. Although the model integrates the functionalities of IWIM into a concrete data-driven model (namely Linda), the cited papers show that IWIM can be embedded in other shared dataspace models in a similar way as for IWIM-Linda.

[1] A logic tuple space is a collection of first-order unitary clauses, identified by a ground term. This space can be used both as a communication device and as a knowledge base.

The ECM coordination model, presented in this book, is also a hybrid model, because it integrates shared dataspace functionalities in a process-oriented view. But before to explain ECM and its instances in the next chapters, we discuss which prerequisites we have considered for our coordination model and language to be used for a simulated embodied autonomous agents' system.

3.6 Prerequisites for a Coordination Model and Language

Having motivated in section 2.6.3 the use of a coordination model and language for designing MASs and for mapping them on a concurrent and distributed architecture, we must ask ourselves which prerequisites a coordination model and language should fulfill, which properties they must have. Our design decisions have been guided by the class of MASs that we want to support, that is, by systems composed of simulated embodied agents (see section 2.5). This does not mean that our coordination model and language cannot be applied to other types of MASs: chapter 9 indeed discusses the implementation of an application belonging to another class of MASs.

We want therefore the following prerequisites:

- Any agent grouping mechanism that encapsulates a set of agents should be supported, allowing local communication and a structuring mechanism for the environment. Furthermore, means to describe nested environments should also be offered.
- The second point, which concerns the representation of agents, can be summarized by three aspects:
  Firstly, agents should be considered as black boxes. They should encapsulate data and behaviour that remain non-accessible for other entities, run concurrently from one another, and act in a pro-active fashion, i.e. they must have the initiative of their action. In this way, the possibility of a basic operational autonomy should be ensured.
  Secondly, we must provide an agent with basic means to access its surroundings. For this reason, an agent should have well-defined interfaces or boundaries with the outside world, through which it can communicate (we name them ports). These boundaries are precisely useful for supporting sensors and effectors.
  Thirdly, we want agents to communicate anonymously through these boundaries, because anonymous communication allows to view communication as an action in the environment and to abstract it from the environment.
- Concerning the means of communication for an agent, we want to support basic communication paradigms, namely peer-to-peer, group and generative communication. In this way, we insure private channels between agents

and provide accesses to blackboards. These latter allow interacting agents to be decoupled in time and space, which is a strong advantage.
The setup of the communication means should be realized in a decentralized manner. No mechanisms should be at disposition to bind explicitly agent boundaries with one another, i.e. to create explicitly connections between these boundaries.
The coordination laws ruling every communication should be clearly defined.

- Dynamicity is a key element: agent creation/destruction and changing communication means should be supported at runtime. The semantics of freed connections therefore has to be carefully specified. Again, this dynamicity should be supported in a decentralized manner: as we are interested in emergent collective behavior properties, a central control over the interactions should be avoided.
- We should use few mechanisms, so that the designer of an application really concentrates on coordination issues.
  Object-oriented technology is especially suited at implementing MASs, basically because agents are encapsulated entities. This is the reason why our coordination language should be incorporated in an existing object-oriented language. Moreover, the coordination language is intended to project a multi-agent application onto a concurrent and distributed architecture. This means that it must exploit concurrency and distribution.

STL [Kro97], [KCD+98], [SKCH98], [KCDS98] (reviewed in chapter 5) has been proposed as a coordination language for multi-threaded applications on a cluster of (UNIX) workstations. We considered the philosophy of STL as presenting high potentials for implementing MASs in the sense of the proposed prerequisites. However, the evaluation of STL showed us several drawbacks with respect to our target. For this reason and in a desire of continuity in the research, we decided to abstract STL's basic ideas and defined, together with colleagues from the PAI group[2], the general coordination model ECM (presented in the next chapter) [SCKH98], [KCDS98], [SCH99]. We then proposed STL++ [SCKH98], [SCH99], [SCK+99] (presented in chapter 6) as a new instance of ECM integrating our requirements in order to support the implementation of our target applications.

The next chapters thoroughly present the coordination model ECM and its instances, and more specifically STL++.

[2] Parallelism and Artificial Intelligence Group of the Computer Science department at the University of Fribourg in Switzerland.

4. The ECM Coordination Model

4.1 Introduction

In this chapter, we present the ECM[1] coordination model [SCKH98], [KCDS98], [SCH99], which proposes coordination abstractions that are orthogonal to computation. ECM is a general model of the coordination languages STL[2] [Kro97], [KCD+98], [SKCH98], [KCDS98], STL++ [SCKH98], [SCH99], [SCK+99] and AGENT&CO. It includes the following design characteristics:

- The model restricts features to the *minimum*, in order to concentrate on coordination duties only. So, only five abstractions have been proposed.
- The model supports only *anonymous communication*; this means that the entities communicate anonymously through communication means without any knowledge of the identity of the communication partners. Furthermore, entities communicate through well-defined ports of communication, which are openings to their surrounding. This has several strong advantages:
  - the programmer can treat entities as black boxes and abandon dependencies coming from the identification of communication partners;
  - this forces the designer to define clean interfaces and it helps the reusability of code.
- The flows of communication are automatically established, according to the state of the communication interfaces. *No coordinator* is in charge of binding ports in order to establish connections; this is done implicitly by the system.
- *Modularity* is a key element of ECM, in the sense that the model supports encapsulation and nested blocks.
- ECM introduces a preliminary level of *privacy*, because connections are private means of communications: only participant processes may read or write information on these connections. However, there is no support for stronger security such as encryption or identification of communication partners.

[1] **E**ncapsulation **C**oordination **M**odel.
[2] **S**imple **T**hread **L**anguage.

M. Schumacher: Objective Coordination in MAS Engineering, LNAI 2039, pp. 53–59, 2001.

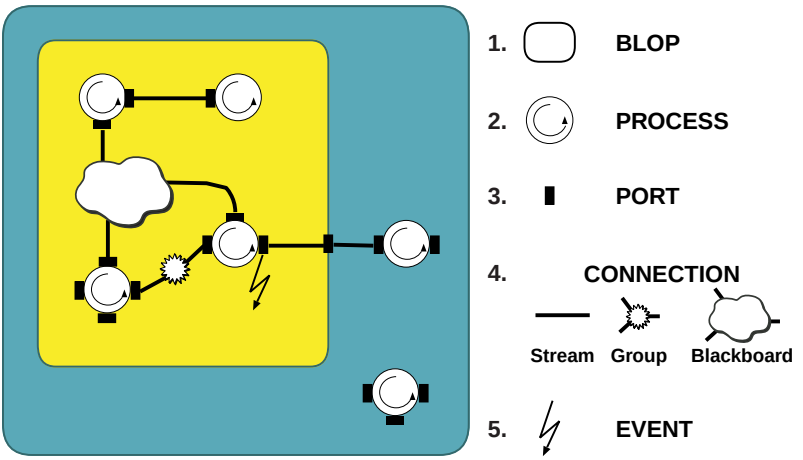

Fig. 4.1. The ECM coordination model.

Figure 4.1 gives a first overview of the programming metaphor used in ECM. The model uses an encapsulation mechanism as its primary abstraction (referred to as blops[3]), offering structured separate name spaces which can be hierarchically organized. Within them, active entities communicate anonymously within and/or across blops through connections, established by the matching of the communication interfaces (the so-called *ports*) of these entities.

Therefore, ECM consists of five building blocks:

1. *Blops*, as an abstraction and modularization mechanism for a group of processes and ports;
2. *Processes*, as a representation of active entities;
3. *Ports*, as the interface of processes/blops to the external world;
4. *Connections*, as a representation of connected ports.
5. *Events*, as a mechanism to support dynamicity (creation of new processes or blops);

According to the general characteristics of what makes up a coordination model and corresponding coordination language (see section 3.2.2), these elements are classified in the following way:

– The *Coordination Entities* of ECM are the processes;
– There are two types of *Coordination Media* in ECM: ports and connections which enable coordination, and blops, the repository in which coordination takes place;
– The *Coordination Laws* are defined through the semantics of the ECM events, the creation of the media, the operations on the media, and the

[3] This is a term STL has coined.

state changes of the media (as a result of operations on them or of state changes of other media).

Designing an application using ECM consists of creating a hierarchy of blops in which several processes run. These processes communicate and coordinate themselves via events and connections. Ports serve as the communication endpoints for connections which result in pairs of matched ports. The next sections explain step by step each ECM key element[4].

4.2 Blop

A blop is a mechanism to *encapsulate* a set of objects. Objects residing in a blop are by default only visible within their "home" blop. Blops have the same interface as processes, a (possibly empty) set of ports, and can be hierarchically structured. Blops serve as a separate matching name space (see section 4.5) for port objects, processes, and subordinated blops as well as an encapsulation mechanism for events.

4.3 Process

A process in ECM is a typed object with a (possibly empty) set of ports. Processes in the ECM model do not know any kind of process identification; instead a black box model is used. A process does not have to care about which process information will be transmitted to or received from.

Process creation and termination are not part of the ECM model and are to be specified in the instance of the model.

4.4 Ports and Connections

Ports are the interface of processes and blops to establish connections to other processes/blops. They can be considered as a gateway between the inner side and the outer side of their owner (port or blop).

4.4.1 Port Features

Each ECM language binding must specify a minimal set of *port features*, each of which describes a port characteristic. The *communication feature* is mandatory for every ECM instance and must support the following communication paradigms:

[4] Relating to the prerequisites presented in section 3.6, section 6.9.2 discusses the use of the key elements of ECM/STL++ for MASs.

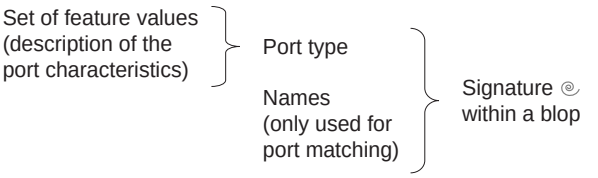

Fig. 4.2. Signature of a port.

- *point-to-point stream communication* (with message-passing semantics): information transits from one entity to another through a private channel; note that the communication partners are not identified, because ECM only supports anonymous communication;
- *closed group communication*: only the members of the group can receive and send information with multi-cast semantics;
- and *blackboard communication*, which allows generative communication.

Additional features are available to refine the semantics of the communication between ports. The ECM coordination model does not specify them, because they are defined by each ECM instance.

The set of feature values of a port defines its *type*. A port is created with one or several *names*. Names are not to be used for identification, but for matching purposes. The names and type of a port are referred to as its *signature*. Figure 4.2 summarizes these notions.

4.4.2 Connections

Connections are established automatically as the result of the matching of ports. Resulting from the basic port semantics, the following generic types of connections are possible (see figure 4.3):

- *Point-to-point Stream*: `1:1, 1:n, n:1` and `n:m` communication patterns are possible;
- *Group*: messages are multicast to all members of the group. A closed group semantics is used, i.e. processes must be members of the group in order to distribute or receive information in it;
- *Blackboard*: messages are placed on a blackboard used by several processes; they are persistent and can be retrieved more than once in a sequence defined by the processes.

4.5 Port Matching

The matching of ports is defined as a relationship between port signatures. Four general conditions must be fulfilled for two ports to match:

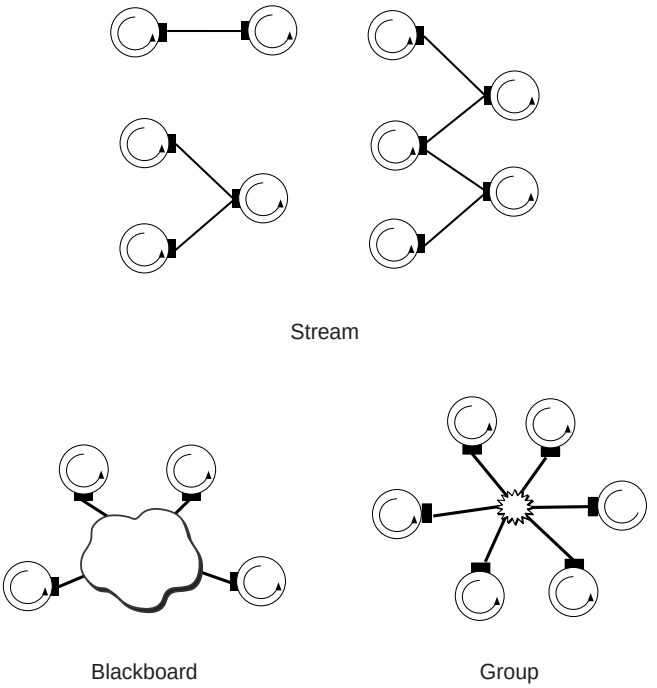

Fig. 4.3. ECM connections.

1. both must share at least a common name;
2. both must belong to the same level of abstraction;
3. both must belong to different objects (process or blop);
4. both types must be compatible: a compliance relationship must be defined for every feature in the ECM instance[5]. Regarding the communication feature, ECM requires that both ports have the same communication paradigm.

The second matching condition has a particularity in what concerns blop ports. As a blop port is a gateway between the interior and the exterior of a blop, it may be connected both on the inner side and the outer side; thus, with regards to this condition, such a port is considered to belong both to the inner abstraction level (the owner blop) and the outer abstraction level (the father blop). This means that the matching is independently realized on each side in order to establish respective connections.

As we have already mentioned, the matching of ports is automatically established. This means that there exists no language construct to bind ports in order to establish a connection: the matching is therefore realized implicitly.

[5] Section 6.4 presents the features used for STL++ and the compliance relationship for each of them.

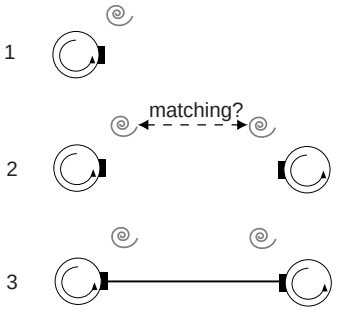

Fig. 4.4. ECM port matching (signatures are indicated as a spiral).

Conceptually, the matching is established in the following manner (see figure 4.4). If two ports are created by their respective entity within the same blop, then their signatures are tested for compliance: if the two signatures have a name in common and their types are compatible, then a connection is established with a semantics depending on the port types.

4.6 Events

Events can be attached to conditions on ports of blops or processes. Conditions check the port state on which they are attached, determining when the event will be triggered in the blop. Therefore, when a condition is fulfilled, the corresponding event is executed; this means that the event runs an action to which it was assigned. An event is not to be understood as a control message, but as a fired action[6]. Typically, it creates new processes or blops.

Each language-binding must determine its proper conditions (see section 6.6 for STL++ condition definitions). Condition checking is implementation dependent.

4.7 ECM Instances

ECM is a general model of coordination in the sense that it defines a frame that must further be specified by its instances. For example, the model does not impose which port features must be supported by the language binding. One exception exist for the communication feature. But also in this case, the semantics of each arisen connection must be specified.

[6] ECM does not define a broadcast mechanism, because the model wants to restrict communication through ports and connections. However, it is possible to simulate a restricted broadcast within a blop by attaching a default group port with a common name to each process in the blop.

Figure 4.5 summarizes the three existing instances of ECM:

- STL, which has been the subject of another research study [Kro97], has been fully implemented as a separate language that must be used with a computation language implementing the coordinated processes. STL is applied to multi-threaded applications in the context of network-based concurrent computing.
- STL++, which is the main subject of this book, is an object-oriented instance of ECM that follows the prerequisites of chapter 3 for our target class of multi-agent systems. It has a single language approach, i.e. the language is implemented as a library extended by few macros. The object-oriented integration is a target design issue: every abstraction of ECM is described by a class, and the primitives on these abstractions become member functions except some creation macros. Furthermore, STL++ emphasizes dynamicity and uniformity in the realization of the ECM constructs.
- AGENT&CO tries to enhance STL++ in two ways. Firstly, AGENT&CO slightly extends ECM in order to support mobility of active entities over blops. Secondly, the language is implemented in JAVA over an object request broker architecture [OMG97b] in order to enhance portability.

These three ECM instances are presented in the next chapters. Chapter 5 presents the coordination language STL. After that, chapter 6 is devoted to a thorough description of our coordination language STL++, accompanied by an overview of its implementation. AGENT&CO is introduced in chapter 7.

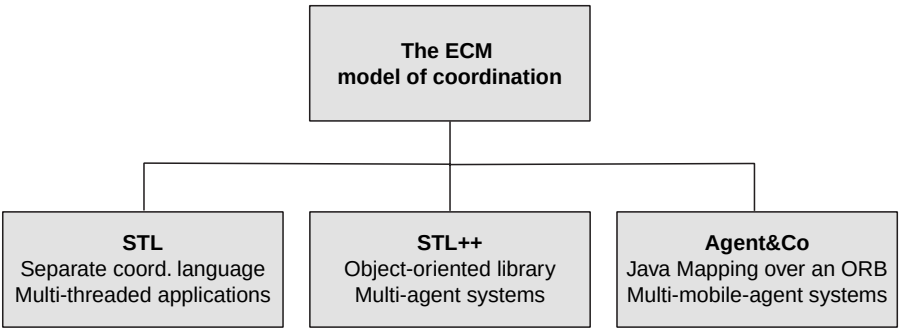

Fig. 4.5. The ECM instances with their implementation and target applications.

5. The STL Coordination Language

5.1 Introduction

STL[1] [Kro97], [KCD+98], [SKCH98], [KCDS98] is applied to multi-threaded applications on a LAN of UNIX workstations. STL materializes the separation of concern as it uses a separate language exclusively reserved for coordination purposes; it however provides some primitives which are used in a computation language to express interactions between the entities. The implementation of STL is based on PT-PVM [KHS96], a library providing message passing and process management facilities at thread and process level for a cluster of workstations. In particular, blops are implemented as heavy-weight UNIX processes, and processes are realized as light-weight processes (threads).

An application written using STL is composed of two parts (see figure 5.1 as an illustration):

- The *computation part* implemented in an ordinary computational language (file `app-func.c`). The implementation supports the C programming language [KR78]: every process is implemented with a function that is launched as a thread.
- The *coordination part* implemented in STL (file `app.stl`).

From the STL code (`app.stl`), the STL-compiler produces pure C code (`app.c`), which uses the PT-PVM communication API; the generated C file (`app.c`) together with the computation files (`app-func.c`) must then be compiled and linked together with the PT-PVM and STL runtime libraries, in order to generate the executable (`app`).

As already stated, STL implements only coordination duties of processes implemented in a computation language (namely C). However, several primitives have been implemented to be used in the computational language. They are necessary to interact with the STL coordination elements. The set of primitives includes the following families: i) Operations to dynamically create ports, referred to as dynamic ports; note that the port type definition can only be done in the coordination language; ii) Operations for process manage-

[1] **S**imple **T**hread **L**anguage.

M. Schumacher: Objective Coordination in MAS Engineering, LNAI 2039, pp. 61–67, 2001.

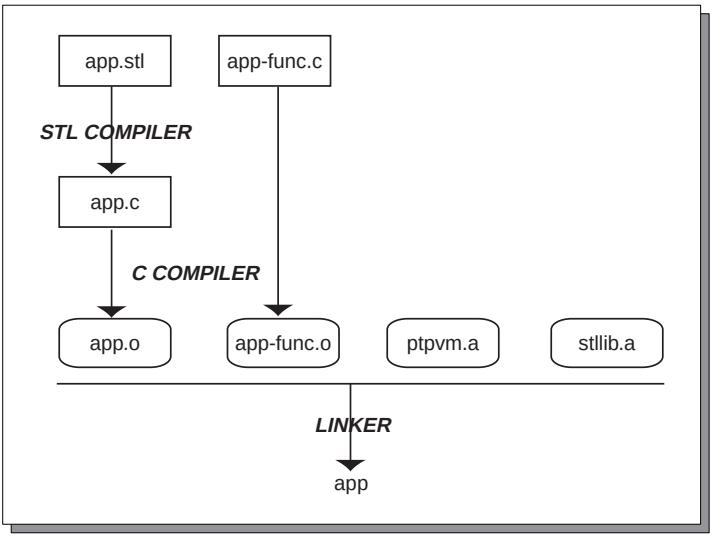

Fig. 5.1. Programming environment of STL.

ment, namely for dynamic creation of processes from within the computation language; iii) Methods to transfer data from and to ports.

Figure 5.2 gives a first impression of what STL code looks like. In a default `world` blop (line 1), which is a root blop encompassing every other entity, a hierarchy of processes and blops are defined (process `the_process` and blop `the_blop` at lines 3 and 10) and then created (lines 14 and 16). This shows that the hierarchical structure is used yet at definition level, because a blop is always defined within another blop.

```
1  blop world {
2
3    process the_process {                    // Process definition
4      P2Pin port1 <"INPUT">;
5      BB port2 <"BB">;                       // with its ports
6      ...
7      func fp;                               // Thread entry point
8    }
9    ...                                      // More processes
10   blop the_blop {
11     ...                                    // A new blop
12   }
13   ...                                      // More blops
14   create process the_process p1;           // Create processes
15   ...
16   create blop the_blop b1();               // Create blops
17   ...
18 }                                          // End of blop world
```

Fig. 5.2. An STL code example.

5.2 Blops

A blop is defined with a specific name; this name is used by an STL code to create new instances of blops. It is possible to create blops on specific computers, or to distribute them transparently on a cluster of workstations (which is done by default if no machine name is indicated).

As blops are hierarchically defined, their instantiation results in a recursive creation of all enclosed blops and static processes. At creation, an identifier is assigned to a blop; it is used only for referring to the ports of the newly created blop within its creator (mostly to attach events to such ports) and is not used by other blops.

Blop declaration and invocation are illustrated in figure 5.3. Two blops are defined: `world` and `sieve`. Using `create blop`, a blop is created somewhere on the cluster (see line 9), assigning to it the name `s`. Lines 5 and 6 (port definitions) are explained later.

```
1  blop world {
2    ...
3    blop sieve {                  // Declaration blop sieve
4                                  // Two ports: types and names
5      Group a <"CONNECTOR">;
6      P2Pout b <"RESULT">;
7    }
8    ...
9    create blop sieve s();        // Create blop
10 }
```

Fig. 5.3. Blop declaration and invocation in STL.

5.3 Processes

STL distinguishes between *static* and *dynamic* processes. Static processes are defined in blop declarations and created through the instantiation of the process object inside the parent blop (by its creation). Dynamic processes can be activated in the body of an event or within the computation language using the primitive `createProcess`.

In STL, process creation assigns an identifier to the process object. As for blops, it is only used for referring to the ports of the newly created process within its creator, mostly in order to attach events.

The introduction of a facility to create new processes in the computation code comes from the necessity of dynamicity. In fact, the reduction of process creation at blop instantiation and event triggering would substantially reduce the expressivity of STL. This constitutes, however, a compromise regarding the goal of STL to totally separate stilistically coordination from computation, i.e. at code level.

Using the keyword `func`, the definition of a process must indicate a function name, which constitutes the thread entry point of this process. Thus, this function is implemented in the computation language, using as arguments the static ports (see next section).

The termination of a process is issued when the corresponding function terminates; thus the process disappears from the blop. STL does not treat the case of pending connections of the terminated process; this may introduce important inconsistencies. Chapter 6 will show how STL++ tries to solve the semantics of process termination and more generally port termination.

Example 5.4 shows how a `worker` process is declared (lines 2 to 6) and created (line 7).

```
1 // Process type worker
2 process worker {
3   P2Pin in <"WORK">          // input port
4   P2Pout res <"RESULT">      // output port
5   func worker;               // Thread entry point
6 }
7 create process worker w;     // An instance
```

Fig. 5.4. Process declaration and invocation in STL.

5.4 Ports and Connections

Table 5.5 gives an overview of STL's port features[2]. By combining them, a user can define in STL several port types. By default, some built-in types are provided by STL. Their respective feature values are summarized in table 5.6. The semantics of the resulting connections directly follows the semantics of the matching ports.

Table 5.5. STL features.

Feature	Explanation	Possible Values
Communication	Communication paradigm	`stream`, `blackboard`, or `group`
Saturation	Amount of ports that may connect	Integers or * for infinity
Capacity	Capacity of the port in data items	Integers of * for infinity
Synchronization	Message passing semantics	Integers of * for infinity
Orientation	Direction of data flow	`in`, `out`, `inout`

The predefined ports encompass the following:

`P2P`

This port type realizes an asynchronous stream port. Sending data on this port is non-blocking; getting data is blocking. The storage capacity is

[2] Originally, STL denominates ECM's features as *attributes*. In order to have a uniformity in our terminology, we keep the term *feature*.

Table 5.6. STL's built-in ports and a user defined port `MyPort` with corresponding port feature values.

Feature	`P2P`	`BB`	`Group`	`MyPort`
Communication	`stream`	`blackboard`	`group`	stream
Saturation	`1`	*	*	`5`
Capacity	*	*	*	`12`
Synchronization	`async`	`async`	`async`	`async`
Orientation	`inout`	`inout`	`inout`	`in`

set to infinite (symbolized by *), and the saturation limited to one other port. By varying the orientation feature's values, the following types are achieved: an output port (`P2Pout`), an input port (`P2Pin`), or both (`P2P`).

`Group`
: With this port, the group mechanism of ECM is realized. Every send operation is based on multicast; the message items will always be transferred to all members of the group, and only to them.

`BB`
: The `BB` implements a blackboard port with a blackboard semantics for the resulting connection. The messages can be stored and retrieved using a symbolic name and a tag.

As depicted in table 5.6, the user can define in STL new port types. For instance, one can define a `1:n` point-to-point connection by setting the saturation feature to `n`. For the concrete example of port type `MyPort` from table 5.6, figure 5.7 shows the necessary code for the port definition. Synchronous communication may also be achieved by setting the `communication` feature to `sync`. Posting data on such a `1:n` port will then block until all the `n` processes have connected to the port and read the data.

```
1 port MyPort {
2     communication = stream;
3     saturation = 5;
4     capacity = 12;
5     synchronization = async;
6     orientation = in;
7 }
```

Fig. 5.7. Defining a port in STL.

STL's port semantics presents several ambiguities. The first ambiguity results from distinguishing two kinds of ports: on one hand *static* ports as statically created in the coordination language; and on the other hand *dynamic* ports as created at runtime in the computation language. Figure 5.4 shows the example of a static port creation: two asynchronous stream ports are created respectively with "`WORK`" (line 3) and "`RESULT`" (line 4) as names. This distinction between static and dynamic ports introduces the following

semantics difficulties: i) matching between a static port and dynamic port is not allowed; ii) events can only be attached to static ports.

If one considers STL's port features in more detail, one will see that some dependencies exist between them. For instance, the `synchronization` feature can override the `capacity` feature; the reason is obvious: synchronous communication implies a capacity of zero.

Another problem of STL is the following: the language does not consider direct deletion of ports. But this may, for instance, implicitly happen when a process terminates; automatically, all ports must terminate. This restriction can introduce semantic inconsistencies. Consider the case of "free" connections within a blop. They will not be reused to be re-connected to free ports: data may thus be lost.

5.5 Port Matching

The matching in STL follows the basic matching conditions of ECM (see section 4.5). Furthermore, for two ports to match, their communication and synchronization features must be equal, both ports must not be saturated, and their orientation features equal (in the case of `inout`) or contradictory (in the case of `in` or `out`).

5.6 Events

ECM events are realized with event handlers inside a blop. An event handler is only defined and used in STL code; there is no primitive defined in the computation language that can treat events.

Figure 5.8 shows how an event is typically defined and used. Lines 1 to 4 define the event `new_worker`. This event will create a new process `worker` and bind recursively to its outgoing port `out` the same event, leading to an infinite recursion.

```
1  event new_worker() {
2    create process worker new;
3    when unbound(new.out) then new_worker();
4  }
5                          // Process type worker
6  process worker {
7    P2Pin in <"WORK">     // in port
8    P2Pout out <"WORK">   // out port
9    func worker;          // Thread entry pt
10 }
11 create process worker w;
12                         // Attach event to port
13 when unbound(w.out) then new_worker();
```

Fig. 5.8. An example of event handling in STL.

After an event has been triggered, it must be re-installed in order to be re-launched. The re-installation is often realized by the same event routine that is currently processed.

STL has defined several conditions that test port states; they are checked every time data flows through the corresponding port or a process accesses the port. An example is the `unbound()` condition that returns `true` if the port is not yet matched to all its potential communication partners (see figure 5.8, line 13).

Further details on STL can be principally found in [Kro97], and also in [SKCH98] and [KCDS98].

6. The STL++ Coordination Language

6.1 Introduction

With the experience of STL, it turned out that in ECM the separation of *code* cannot always be easily maintained. Although the blackbox process model of ECM is a good attempt to separate coordination and computation code, dynamic properties proved to be difficult to be expressed in a separate language.

This consideration led us to initiate the development of our new coordination language called STL++ [SCKH98] [SCH99], [SCK+99]. Starting from the experience acquired with STL, a single language approach has been adopted: STL++ implements the conceptual model of ECM by embedding it in a given object-oriented language (namely C++ [Str97]) with coordination primitives realized in a library.

This chapter is organized as follows. In this introduction, we expose the principal design decisions of STL++ and give an overview of an STL++ application. Sections 6.2 to 6.6 give a semi-formal presentation of STL++. Section 6.7 discusses a tutorial example implemented in STL++. Section 6.8 sketches the main characteristics of a prototype implementation of STL++. Section 6.9 concludes the chapter with a discussion. Appendix A systematically presents the core interface of STL++.

6.1.1 Design Decisions

In the following paragraphs, we want to explain the reasons that motivated the definition of our new ECM instance. This is done by showing which design decisions have been taken. It is straightforward that, being a mapping of the coordination model ECM, STL++ directly follows the design characteristics described in section 4.1. But, in addition, the language binding has the following motives and conceptual decisions:

– *Dynamicity.* The coordination language should support dynamicity as a key element of our requirements (see section 3.6): the system should be able to dynamically create and connect new services. In order to achieve this, blops, processes and ports should be created dynamically and the connection patterns should change during program execution. Processes must also

M. Schumacher: Objective Coordination in MAS Engineering, LNAI 2039, pp. 65–96, 2001.

have the possibility to delete ports (bound connections may terminate), whereby new ports can connect to existing ports or free connections.

– *Single language approach.* As already said, our experience showed us that an approach based on a separate coordination language makes compromises with respect to its goal design, in the sense that it introduces several facilities in the computation language in order to achieve the maximum flexibility in the interactions with coordination duties, especially in what concerns dynamic aspects.
  This is reflected in STL, where several coordination elements are present in the computation language to provide dynamic coordination facilities. These elements include the definition of dynamic ports, few primitives for creating dynamically these ports and launching also dynamically declared processes, and operations for transferring data from and to the ports. Hence, in ECM, dynamic properties cannot be totally separated from the actual program code.
  Therefore, in STL, in addition to the fact that each process and its ports must be declared both in the computation language and in the separate coordination language, a duplication of code concerning coordination duties is often inevitable. As a result, this duplication may introduce some difficulties in managing an application.
  For these reasons, STL++ is proposed rather as an extension of an existing computation language, than as a definition of a separate one. In this way, we achieve a uniformity in the programming methodology while maintaining the expressivity of the host programming language, for instance that of typing and control structures.
– *Object-orientation.* Object-orientation is a key design issue, in the sense that the objective is an object-oriented coordination language. In STL++, blops, processes, ports and connections become objects, and the different primitives become member functions. An STL++ program is then a collection of classes and their instantiations.
  Strong typing and class libraries allow an increase of the expressivity. Modularity and reuse of code are also important requirements for STL++ and are automatically applicable in the object-orientation paradigm. In this way, we make blops and processes to be parameterizable, which is not the case in STL.
  ECM processes and blops are implemented as active objects. But object-orientation for these entities is restricted to the ECM model: the attributes of the active objects (blops and processes) are not accessible by other blops and processes (not even ports), in order to limit communication through the ports to the established connections.
– *Simplicity* and *uniformity.* STL++ should be restricted to the minimum of constructs and follow a uniformity in their realization. For instance, there is no more distinction between static and dynamic ports or static and dynamic processes (as is the case in STL), because this introduces semantic

difficulties. Moreover, message manipulations are handled by ports: a user does not need to buffer data before passing them to a port, as it is the case in STL.

– *Semantics.* The semantics of each ECM construct should be precisely defined in order to avoid any kind of ambiguity. Two issues are particularly important in that respect. Firstly, the matching mechanism between several ports must be cleanly specified in order to avoid each confusion. Secondly, as a consequence of enhanced dynamicity, the mechanisms and the semantics of the termination of ports and processes must be studied in order to keep consistency of the freed connections and their respective pending data. This is only possible if the consistency state of each type of connection is analyzed.

6.1.2 An Overview

Our systematic description of each STL++ element is illustrated by a simple example, namely the sieve of Eratosthenes, which searches for prime numbers. In [Kro97], the same example is described using STL. In order to allow a comparison, we expose the STL code in appendix B.

We briefly recall how the sieve works and present the overall construction of an STL++ application.

The Sieve of Eratosthenes. Figure 6.1 pictures the organization of the sieve. A `source` agent[1] produces integers. A filtering pipeline is gradually constructed with filter agents (`sieve`) by attaching, on unbound ports, events that create new `sieve` agents; the used event condition tests if the port is already bound or not. Each `sieve` agent is responsible for one prime number by filtering its multiples (the primes are ascending in the pipeline). When it cannot divide a number, it writes it on its port bound to the pipeline. If this port is not bound (which means that the new number is a prime), an event creates a new `sieve`. The newly created sieve agent receives its prime, writes it on a result port bound to a connection with the source agent, and begins its filtering job.

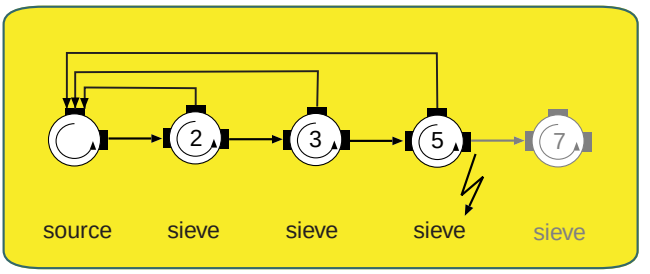

Fig. 6.1. The sieve of Eratosthenes.

[1] ECM processes are named agents in STL++.

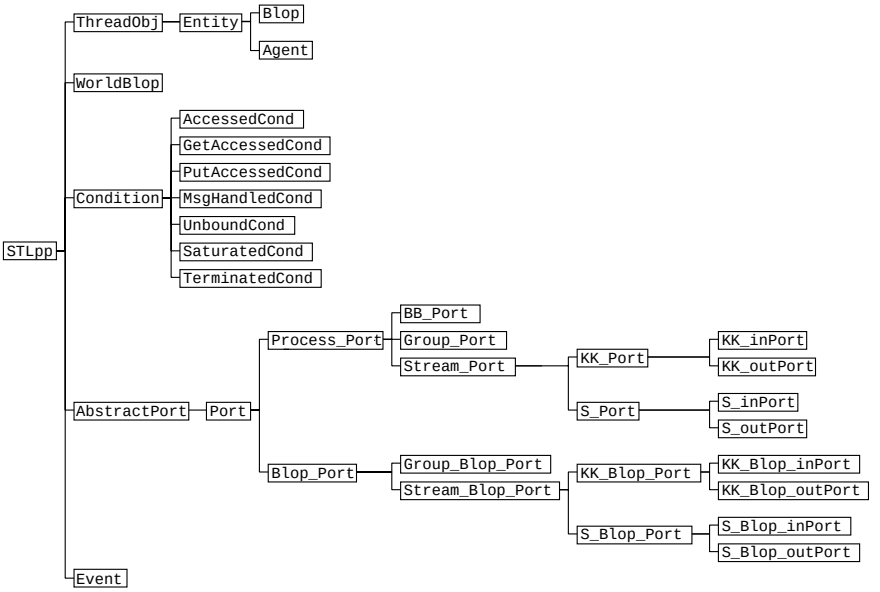

Fig. 6.2. STL++ user classes.

Overall Construction of an STL++ Application. An STL++ application is a set of classes that inherit from the base classes of the user library depicted in figure 6.2. Several methods must then be re-implemented. For the sieve example, figure 6.3 shows the classes re-implemented.

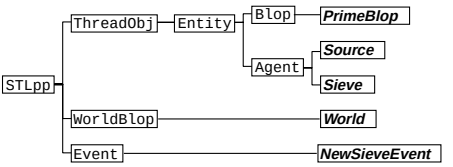

Fig. 6.3. Classes of the example of the sieve of Eratosthenes.

First, the `main` procedure must initialize the system by calling `STLppInit()`. Then an instance of the main class must be created; this main class must inherit from `WorldBlop`, which is the default blop containing all other entities. Last, `STLppTerminate()` is called in order to wait for every activity in the system to terminate; the system is then shut down.

Every blop and agent class must be pre-declared using the STL++ macro `declareRunnable`. This macro allows to embed the instances of the respective classes in a thread. The implementation must then define the blop and agent

classes as a blop or agent with their respective constructor types; this is done respectively with the macro `defineBlop` and `defineAgent`.

An application is launched on a distributed network of UNIX workstations: a number of machines must be indicated at any program start. During program execution, blops are transparently distributed over the cluster of workstations. Agents are always created locally, i.e. on the same node as its creator.

6.2 Blops

A blop class inherits from the base class `Blop` and has to re-implement its `start` method; this method contains the description of the blop. In it, the user must first call `Blop::start()` in order to initialize the blop.

In STL++, new blops can be dynamically created at runtime. This creation, which can be done by other blops, agents or events, results in the initialization of all enclosed blops, ports and agents. Actually, the role of a blop is to handle all its enclosed entities. Furthermore, as ECM asks for, a blop can also create its own ports; these ports are then gateways binding the external with the internal sides of the blop.

The blop creation is done using the macro `createBlop`. The user must indicate to this macro the name of a blop class to create. No reference is given back. Note also that the blop constructor can be parameterized as for normal classes.

In figure 6.4 can be found for our example the declaration of a blop (class `PrimeBlop`) and the re-implementation of its `start` method.

```
1  // Class PrimeBlop declaration
2  class PrimeBlop : public Blop {
3   public:
4    PrimeBlop();
5    ~PrimeBlop();
6    void start();
7  };
8
9  // start() method implementation
10 void PrimeBlop::start() {
11   Blop::start();          // Initialize blop
12   createAgent(Source);    // Create source agent
13   stopMe();               // Wait for every entity
14 }                         // to stop, and terminate
```

Fig. 6.4. A blop class and the re-implementation of its `start` method for the problem of the Sieve of Eratosthenes.

6.3 Processes

In STL++, ECM processes are named agents, because they are intended at supporting the implementation of autonomous agents. Agent classes inherit from the base class `Agent` and have to re-implement the `start()` method. Agents can then be created dynamically by blops, through events or by other agents.

In order to create an agent, the macro `createAgent` must be used, indicating the name of the corresponding agent class. No reference is given back; there is no identification mechanism between agents: agents only communicate through their ports.

In the `start` method, an agent is initialized with `Agent::start()`. As defined in the ECM model, to communicate, agents dynamically create ports of predefined types through instantiation of a port C++-template class and call communication methods on them.

In order to terminate, an agent has to call the method `stopMe`, which terminates the agent with its ports and data. The pending connections remain as long as they are bound on the other side. A connection that is free on one side can be re-connected to other ports (see section 6.5). In that case, the matching mechanism will consider free connections in priority to other ports.

Regarding the implementation, an `Agent` object is embedded in a light-weight process (thread), which is in charge of creating an instance of the agent class and launching the `start` method.

In figures 6.5 and 6.6 can respectively be found the declaration of the agent classes `Source` and `Sieve`, and the re-implementation of their `start` methods. Although several notions are explained in the next sections, we briefly explain the `start` methods of these two classes.

After initialization, the source agent creates three ports: `outP` (line 18) for feeding the pipeline with integers, `inP` (line 19) for sampling found primes, and `terminationP` (line 20) for getting the number of primes found, thus entering the termination phase. An event of class `NewSieveEvent` is created (line 22) and attached to `outP` (line 23) in order to create the first sieve agent of the pipeline. Values are then transmitted in order to initialize the first sieve agent (lines 24 and 25). The main loop (lines 26 to 28) feeds the pipeline with integers. When `INT_LIMIT` integers have been produced, a termination message is posted on port `outP` (line 29). This allows to sample the primes on port `inP` (lines 30 to 33) and to terminate (line 34).

The sieve agent creates three ports: `inP` (line 18) for getting the owned prime, the actual number of primes and the integers to filter, `primeP` (line 19) for transmitting the prime to the source agent, and `outP` for forwarding integers that cannot be divided (line 20). The sieve first gets its prime (line 21) and the actual number of primes (line 22), and transmits its prime to the source agent (line 23). Then an event of class `NewSieveEvent` is created (line 24) and attached to `outP` (line 25) so that a new sieve agent can be created when a new prime is found and posted on this port. The main loop tries to

```
1  class Source : public Agent {
2   public:
3    Source();
4    ~Source();
5    void start();
6   private:
7    KK_outPort<int> *outP;
8    KK_inPort<int> *inP, *terminationP;
9    int nextNumber, numberOfPrimes;
10   void archive(int);
11 };
12
13 void Source :: start() {
14   Agent::start();                    // Initialize source agent
15   numberOfPrimes = 1;
16   nextNumber = 2;
17   // Create ports:
18   outP = new KK_outPort<int>(this, nV("INTEGERS"), 1, INF);
19   inP = new KK_inPort<int>(this, nV("PRIMES"), INF, INF);
20   terminationP = new KK_inPort<int>(this,nV("PRIMES_NBR"),1,INF);
21   // This event creates the first element of the pipeline:
22   NewSieveEvent *e = new NewSieveEvent();
23   outP->attachEvent(e);
24   outP->put(nextNumber++);           // Initialization of the
25   outP->put(numberOfPrimes);         //              first sieve
26   while(nextNumber < INT_LIMIT) {    // Main loop
27     outP->put(nextNumber++);         // Feed the pipeline with int
28   }
29   outP->put(-1);                     // Post termination msg
30   for(numberOfPrimes = terminationP->get(); // Enter termination
31       numberOfPrimes > 0; --numberOfPrimes) {
32     archive(inP->get());             // Archive the results
33   }
34   stopMe();                          // Terminate
35 }
```

Fig. 6.5. Declaration of the `Source` agent class and the implementation of its `start` method for the problem of the Sieve of Eratosthenes.

filter incoming integers (lines 27 to 36): if a number is not divisible by the prime (line 29), it is forwarded in the pipeline (line 30). After a termination message has been read, the loop ends. If the sieve is the last in the pipeline (line 37), the termination port `terminationP` is created and used for posting the number of primes to the source agent (lines 38 and 40). Finally the sieve agent terminates (line 42).

6.4 Ports and Connections

6.4.1 Port Features

In accordance with the ECM model, each port is endowed with one or several names and a set of features. STL++ distinguishes *Primary* and *Secondary Features* for ports.

Primary Features, which define the main semantics of a port, and therefore the basic port types, encompass:

```
class Sieve : public Agent {
 public:
  Sieve();
  ~Sieve();
  void start();
 private:
  KK_inPort<int> *inP;
  KK_outPort<int> *primeP, *outP, *terminationP;
  int nextNumber, myPrime, numberOfPrimes;
  bool last;
  void stopMe();
};

void Sieve :: start() {
  Agent::start();              // Initialize sieve agent
  last = true;
  // Create ports:
  inP = new KK_inPort<int>(this, nV("INTEGERS"), 1, INF);
  primeP = new KK_outPort<int>(this, nV("PRIMES"), 1, INF);
  outP = new KK_outPort<int>(this, nV("INTEGERS"), 1, INF);
  myPrime = inP->get();        // Get my prime number
  numberOfPrimes = inP->get();
                               // Get the actual number of primes
  primeP->put(myPrime);        // Tell source agent
  NewSieveEvent *e = new NewSieveEvent(); // Create Event
  outP->attachEvent(e);        // And attach it to outP
  // Main loop (as long as no termination msg)
  for(nextNumber = inP->get();
      nextNumber>0; nextNumber = inP->get()){
    if (nextNumber % myPrime != 0) {
                               // If nextNumber not divisible,
      outP->put(nextNumber); // Post nextNumber on my outP port
      if (last) {              // If I am the last in the pipeline
        outP->put(++numberOfPrimes);
                               // Indicate number of primes
        last = false;          // I am not any more the last
      }
    }
  }
  if (last) {                  // If I am the last
    terminationP =
      new KK_outPort<int>(this, nV("PRIMES_NBR"), 1, INF);
    terminationP->put(numberOfPrimes);
                               // Post nbr of pr. to source
  }
  stopMe();                    // Terminate if primeP is connected
}                              // i.e. if prime has been transmitted.
```

Fig. 6.6. Declaration of the `Sieve` agent class and the implementation of its `start` method for the problem of the Sieve of Eratosthenes.

- *Communication structure*: this feature captures the basic ECM communication paradigms, namely *point-to-point stream*, *group* and *blackboard* communication;
- *Type of message synchronization*: this feature gives to the connection the usual semantics of message passing communication. The two possible semantics are *synchronous* and *asynchronous*.
  When an agent writes on a synchronous port, it blocks until the data items have been read by a consumer process on the connection. If the port is bound to multiple connections, it will block until the echoed data item

has been read on all connections. In the case that the port is not bound, the agent is blocked until a connection is established and data items are read.
For asynchronous communication, the data items written on such a port do not block the producing agent. Nothing can be said about whether data have been read by a consumer process or not.
For asynchronous ports, there is no capacity limitations of ports and connections (like STL does with the *capacity* feature). This means that an agent writing on a non-connected port will not block and will be able to continue to write on such a port; actually the data are stored in a buffer belonging to the port. As soon as a match with another port is achieved, data in the buffer will be forwarded to the new connection.
If the port is nevertheless bound to a connection when data items are written on it, data will be stored in the connection buffer (for `KK-Stream` and `Blackboard` connections) or at the receiver side (for `Group` connections).
- *Orientation*: this feature defines the direction of the data flow over a connection; there are three possible values, namely *in* (in-flowing), *out* (out-flowing) and *inout* (bi-directional).

Secondary Features, which define characteristics for specific types of ports, are:

- *Saturation*: this feature, ranging from 1 to `INF` (infinity) by integer values, defines the number of connections a port can have. This is different to the STL semantics which defines the saturation as the number of ports that can connect to the concerned port (through any number of connections).
The saturation feature has a specific semantics on blop ports. These ports can be seen as having a double saturation, one for the connections inside the blop and another for the connections outside it. The start values of both saturations are equal to the value given at the creation of the port.
Note also that the STL++ blackboard ports have a saturation of 1, in order to simplify their semantics.
- *Lifetime*: this feature, ranging from 1 to `INF` (infinity) by integer values, indicates the number of data units that can pass through a port before it decays (it is not mandatory that the port is bound in order to increment the passed lifetime).
- *Data Type*: this feature defines the type of data authorized to pass through a port. Although every type is possible, the user should not use types including pointers, but basic data types and composed types.

6.4.2 Creation and Destruction of Ports

In order to support dynamicity, STL++ gives facilities for port creation and destruction.

Table 6.7. Basic port types and values of their primary features.

Port Type	Communication	Msg Synchronization	Orientation
Blackboard	`blackboard`	`asynchron`	`inout`
Group	`group`	`asynchron`	`inout`
S-Stream	`stream`	`synchron`	`in` or `out`
KK-Stream	`stream`	`asynchron`	`in` or `out`

In fact, ports can be created at runtime by blops and processes. Additionally, events can make instances of ports. Examples of port creation can be found at figure 6.5 on lines 19 and 20, and at figure 6.6 on lines 19 to 21.

Furthermore, ports can disappear in two ways: they can be destroyed by their creator, or vanish when their lifetime is over. The death of ports has two generic results. If the port is not bound (no connections at all), then all buffered data items are lost. If it is bound to one or more connections, the death of a bound port causes the connections to be freed at the corresponding side. Data items that have been previously written on such a port have been echoed and therefore stored on each connection. Section 6.4.3 explains the semantics of port disappearance for each basic connection type.

6.4.3 Basic Port Types and Their Connections

STL++ currently supports four basic port types, corresponding to the combinations of the primary features as displayed in table 6.7. Further types could have been proposed (for instance synchronous groups). But the proposal covers our basic needs.

To ease the implementation, two base port classes for every port type have been defined, one for agents and one for blops. Port classes are implemented as template classes. The *saturation* and the *lifetime* features are used as constructor arguments, the *data type* feature as the template argument. Moreover, a vector of names must be passed as argument to the constructor of a port.

In this subsection, the semantics of the port types and their corresponding connections are explained:

Blackboard Port Type

The resulting connection, as a result of the matching of ports of this type, has a blackboard semantics. The number of participating ports is unlimited. Messages are persistent objects which can be retrieved using a symbolic name. Moreover, messages are not ordered within the blackboard, which is a multiset.

Blackboard connections are persistent: if all the ports involved in a blackboard connection disappear, the connection still persists in the blop space with all the information it carries, so that new ports can later re-connect to the blackboard and recover the pending information.

To access the blackboard, the following LINDA-like primitives are provided: `put` (non-blocking), `get` (blocking) and `read` (blocking).
At the moment, STL++ does not have an associative mechanism as LINDA. The storing of the data and the access to it is simply done using a string. This point should be enhanced.

Group Port Type
The resulting connection has a closed group semantics; this means that the messages will be sent only to the members of the group. The number of participating ports is unlimited. Each member of the group can multicast asynchronously messages to every participant in the group.
The messages are stored at the receiver side. Thus, if a port in a group disappears, then the sequence of information that has not been read is lost.
The primitives for accessing a group port are: `get` (blocking) and `put` (non-blocking).

S-Stream Port Type
The semantics of a connection resulting from the matching of two S-Stream ports has the semantics of a synchronous channel (like the S-Channels defined in IWIM [Arb96]). In particular this connection is unidirectional. It always results from the matching of contradictory oriented ports, namely a source port (belonging to the producer agent) and a sink port (of the consumer agent).
In contrast to other connections, this connection never contains data, due to its synchronous nature. So the destruction of the producer or the consumer never causes loss of data.
The primitives for accessing the port are `get` (blocking) and `put` (blocking).

KK-Stream Port Type
The semantics of a connection resulting from the matching of two KK-Stream ports (K for keep) is analogous to the asynchronous KK-Channel defined in IWIM [Arb96], with its semantics (see below) when a port disappears from one end of the connection. As for S-Streams, this connection always results from the matching of contradictory oriented ports.
If the connection is broken at its consuming port, the next new matching port will consume all pending data. If the connection is broken at its producing port, the consuming port will be able to continue to consume all data in the connection. If both ports are deleted, the connection disappears with its data. The pending sides will be re-connected in priority.
The primitives for accessing the port are `get` (blocking) and `put` (non-blocking).

STL++ provides also a `getConnNbr` method on a port, thanks to which it is possible at any moment to ask a port for the number of connections to which it is bound.

Table 6.8. Compatibility for features F for two ports P1 and P2.

Feature F	Values	Compatibility
Communication	`blackboard`, `stream`, or `group`	`P1.F` = `P2.F`
Msg. Synchronization	`synchron`, `asynchron`	`P1.F` = `P2.F`
Orientation	`in, out, inout`	(`P1.F` = `in` *and* `P2.F` = `out`) *or* (`P1.F` = `P2.F` = `inout`)
Saturation	`{1,2,...,INF}`	Always compatible
Data Type	Type	`P1.F` = `P2.F`
Lifetime	`{1,2,...,INF}`	Always compatible

6.5 Port Matching

Communication between agents is realized through connections which are the result of matched ports. In accordance with the ECM model, the matching is realized as a relation between port signatures. It is important to see that the matching relationship is not static but dynamic, because it depends on the saturation feature which is determined at runtime, and on the actually available free ports and connections.

Free connections, which result from the destruction of ports, are treated in priority to other ports. This semantics is chosen in order to get the maximum of pending data.

In order to match, ports must comply at name and type levels:

- *Name level.* Two ports match at name level if they share at least one name. A port may have several names; in this case, each name belongs to a different connection;
- *Type level.* For two ports, values of the same feature must be compatible. Table 6.8 gives an overview of the compatibility functions used by the STL++ runtime system.

Furthermore, a port can only match, if it is not saturated.

Stream blop ports have a peculiarity concerning their direction feature. Actually, for the matching rules, such a port can be considered as being composed of two sides: the inner and the outer sides. On the outside of a blop, an out-going stream port is viewed as an "out"-port to which an "in"-port (for instance on an agent) can connect; on the inner of the blop, it is considered as in "in"-port to which an "out"-port can be bound. An in-going blop stream port has exactly the reverse semantics.

By introducing several names for each port, STL++ allows a stream port to be connected to several stream connections, and a group ports to be connected to different group connections (blackboard ports cannot have multiple connections). Data written on such multiple connection ports are echoed on every connection. For stream connections, `1:1`, `1:n`, `n:1` and `n:m` communication patterns can therefore be built. Likewise, several groups can

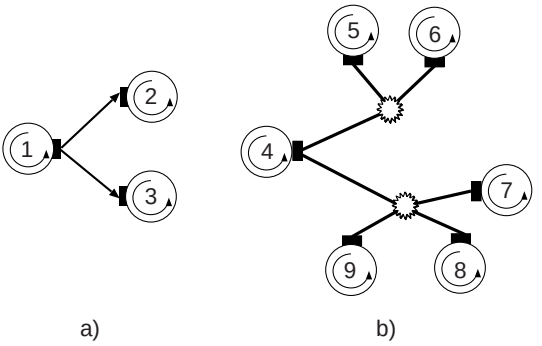

Fig. 6.9. Ports with multiple connections: a) stream b) group.

be connected to a single port. Figure 6.9 shows two examples: a) an output port is connected to two other stream connections; b) a group port is connected to two group connections.

6.6 Events

Event classes inherit from `Event`; the `launch()` method, which defines the acting of the event, must be re-implemented. Events are instantiated with a specific lifetime which determines how many times they can be triggered.

Conditions on ports check the state of the port. They are verified when data flow through the port, or, in the case of the `saturatedCond` condition, when a new connection is realized.

The defined conditions encompass (see table 6.10 for an overview):

`UnboundCond`
: This condition returns `true` if the port is not connected at all, i.e. if the port is not bound to at least one connection[2]. This condition can be elegantly applied to the dynamical construction of pipe-lines. On writing on an unbound port, a new agent can be created by an event; in turn, this new agent has an in-going port that matches to the unbound port of its creator. This scheme is applied to the `sieve` agent in figure 6.6.

`SaturatedCond`
: Returns `true` if the port is saturated, thus if it is bound to all its potential connections. This condition is useful to distribute connections on several ports, in order to avoid bottle-necks. When a port has reached its saturation, it is possible to create a new one in order to tackle future connections.

[2] STL defines this condition with another semantics; the condition is verified if the port has not connected to all its potential connections specified by the saturation.

Table 6.10. Port conditions.

Condition	Explanation
`UnboundCond`	Port not connected.
`SaturatedCond`	Port saturated.
`MsgHandledCond(int n)`	Port has handled `n` messages.
`AccessedCond`	Port accessed.
`PutAccessedCond`	Port accessed with `put` primitive.
`GetAccessedCond`	Port accessed with `get` primitive.
`TerminatedCond`	Port lifetime is over.

`MsgHandledCond(int n)`
: Returns `true` if `n` messages have passed through the port. This condition can be used to control the flow of data through a specific port. Initiatives can be taken if specific thresholds are exceeded.

`AccessedCond`
: Returns `true` whenever the port has been accessed by a primitive.

`PutAccessedCond`
: Returns `true` whenever the port has been accessed by the `put` primitive.

`GetAccessedCond`
: Returns `true` whenever the port has been accessed by the `get` primitive.

`TerminatedCond`
: Returns `true` whenever the lifetime is over. This condition is very useful to indicate when a port should disappear. Consequent actions can be engaged in order to react to this situation, for instance by creating a new port.

Figure 6.11 shows the declaration of an event class and the implementation of its constructor and `launch` method. The constructor uses the `unbound` condition with a lifetime of 1 (line 13). The acting of this event, which is used in our sieve example, creates a new `sieve` agent (line 17). In figure 6.6 can be seen how an event is created (line 24) and attached to the `outP` port (line 25).

6.7 A Tutorial Example

This section is devoted to a practical introduction of STL++ that illustrates the use of the coordination language. This is achieved by presenting a simple tutorial program, the classical Dijkstra's problem of the *Restaurant of Dining Philosophers*. Even if this example program does not belong to our target applications, it helps understanding the mechanisms of STL++. Furthermore, in order to allow a comparison with STL++, appendix C exposes implementations of the same example in LINDA, GAMMA and MANIFOLD, as proposed in [ACH98].

```
1  // Definition
2
3  class NewSieveEvent : public Event {
4   public:
5    NewSieveEvent();
6    ~NewSieveEvent();
7    void launch();
8  };
9
10 // Implementation
11
12 NewSieveEvent :: NewSieveEvent()  // Event creation
13   : Event(new UnboundCond(), 1) {
14 }
15
16 void NewSieveEvent :: launch() {  // Acting of the event
17   createAgent(Sieve);
18 }
```

Fig. 6.11. Declaration of the `NewSieveEvent` event class and the implementation of its constructor and its `launch` method for the problem of the Sieve of Eratosthenes.

6.7.1 The Restaurant of Dining Philosophers

We briefly recall how the problem of the *Restaurant of Dining Philosophers* works.

The restaurant is composed of one table on which philosophers can have a meal. Around the table a fixed number of seats is disposed. Between each seat there is exactly one fork. In order to eat the noodles that are on the middle of the table, a philosopher must pick up two forks, one on the left and the other on the right. When he has finished eating, he puts down both forks, and thinks until he is hungry again.

6.7.2 General Description of the Implementation

We give in figure 6.12 a graphical representation of our solution in STL++ to the problem presented. Five participants to the meal are represented.

A restaurant blop creates a waiter agent that is in charge of running the restaurant. This waiter is responsible for organizing a place for each newly arrived philosopher. In order to simplify, he creates a starting number of philosophers indicated at its creation. After creating them, he decides to collect the money that every client paid for eating; then, he stops in order to close the restaurant.

To show how STL++ constructs are used, the communication means of the example are intentionally enhanced. Concretely, two types of private blackboards appear as a result of matching ports. Firstly, a "meal blackboard" connects the waiter and all philosophers. If there are n philosophers, this blackboard is the repository of $n-1$ tickets used to avoid deadlocks and so to permit the philosophers to pickup their forks. Furthermore this blackboard is the mean to pay. Secondly, a "fork blackboard" is created between each neighbor pair in order to contain a unique fork that is shared by both

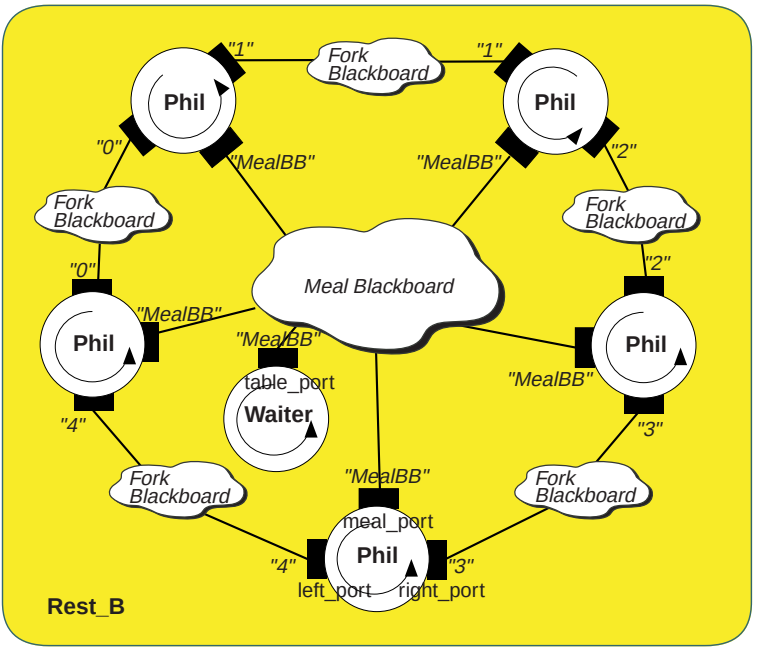

Fig. 6.12. Dining philosophers implemented with STL++.

concerned philosophers. Note that it would have also been possible to use a solution through a single blackboard for controlling the whole interactions. But this would not have exposed the use of several connection means.

Our application encompasses the following classes: `Rest_B`, the restaurant blop class; `Waiter`, the waiter agent class; `Phil`, the philosopher agent class; and `OpenPhilWorld_B`, the world blop class. Their inheritance hierarchy is showed in figure 6.13. The next paragraphs explain each class in the application, except `OpenPhilWorld_B` that simply creates the restaurant blop.

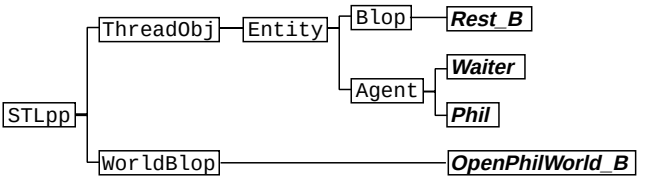

Fig. 6.13. Classes of the dining philosophers program.

6.7.3 The Restaurant Blop and the Waiter Agent

The Restaurant Blop `Rest_B` encompasses every activity in the restaurant. Figure 6.14 shows the class declaration and the implementation of the `start()` method. The blop creates a `Waiter` agent (line 11) indicating it the number of philosophers; after that, the blop waits until every entity has terminated in itself (line 12).

```
1  class Rest_B : public Blop {
2   public:
3    Rest_B();
4    ~Rest_B();
5    void start();
6  };
7
8  void Rest_B :: start() {
9    Blop::start();
10   int nbrOfPhil = 5;
11   createAgent(Waiter, &nbrOfPhil);
12   stopMe();
13 }
```

Fig. 6.14. The `Rest_B` class declaration and the implementation of its `start()` method for the problem of the Dining Philosophers.

As already said, the `Waiter` agent (see figure 6.15) is responsible to run the restaurant. After calling `Agent::start` in its `start` method, the waiter creates a `table_port` port of type `BB_Port` with the name `"MealBB"` and an infinite lifetime (line 18); this port will be used to collect the money. Then the waiter creates the philosophers of class `Phil` (line 22). Finally he accumulates the money through its `table_port` port (lines 25 to 29) and stops (line 30).

6.7.4 The Philosophers

The `Phil` class implements the philosophers (see figure 6.16). At its creation, a philosopher agent is initialized by two integers indicating its right and its left. After calling `Agent::start()` for initializing, its `start` method creates a `meal_port` port of `BB_Port` type with the name `"MealBB"` (line 23). Apart from the first created philosopher, every client puts on this port a meal-ticket used to avoid dead-locks (lines 26 to 29). $n - 1$ tickets are thus available at the beginning. The philosopher creates two other `BB_Port` ports named `right_port` (line 24) and `left_port` (line 25): they are respective accesses to a blackboard which will be the repository of a single fork (each agent puts on its left one fork, see line 30). These two ports are created using the left and right integers as names. In this way, they only match their respective neighbor port.

After this initialization phase, the philosopher calls the `dining` method (line 31), entering a cycle (lines 36 to 39) in which he takes a meal ticket

```
1  class Waiter : public Agent {
2   public:
3    Waiter(int);
4    ~Waiter();
5    void start();
6   private:
7    BB_Port<int>* table_port;
8    int new_creations;
9    int phil_nbr;
10   int total_income;
11   bool continue_p();
12   void rest();
13 };
14
15 void Waiter :: start() {
16   Agent::start();
17   int income;
18   table_port = new BB_Port<int>(this, nV("MealBB"), INF);
19   int i, j;
20   for (i=0;i!=nbrOfPhil;i++) {
21     j = (i+1)%nbrOfPhil;
22     createAgent(Phil, &i, &j);   // Create the philosophers
23   }
24   // Termination: take the whole money
25   income = table_port->get("money");
26   while(income) {
27     total_income += income;
28     income = table_port->get("money");
29   }
30   stopMe();
31 }
```

Fig. 6.15. The `Waiter` class declaration and the implementation of its `start()` method for the problem of the Dining Philosophers.

(`getMealticket` method), picks up its forks (`getForks`), eats (`eat`), puts the forks (`putForks`) and the ticket (`putMealticket`) back, and thinks (`think`). When he is satisfied, he exits the loop in order to leave the restaurant (`stopMe` at line 32). All the methods called in the cycle are explained below.

Getting a ticket (`getMealticket`) is realized through the `table_port` port bound to the blackboard on which the tickets are stored (line 43). If all tickets are used, the agent asking for a ticket will be blocked until one is available.

In order to take the forks (`getForks`), the philosopher gets them on its `left_port` (line 46) and `right_port` (line 47) ports. After having eaten, he puts the forks back (`putForks`) on their respective blackboard (lines 50 and 51) and posts the ticket on its `meal_port` port (line 56 in `putMealticket`). Agents waiting for a fork on the respective connection will be unblocked.

The `stopMe` method of `Phil` is re-implemented in order to integrate the payment of the meal. In fact, the philosopher puts money on its `meal_port` port (lines 60) and, finally, calls its parent method `Agent::stopMe()` in order to terminate (line 61).

```
class Phil : public Agent {
 public:
  Phil(int, int);
  ~Phil();
  void start();
 protected:
  int id, living, myLeft, myRight, aTicket, rFork, lFork;
  BB_Port<int> *meal_port, *right_port, *left_port;
  void getForks();
  void putForks();
  void getMealticket();
  void putMealticket();
  void rest();
  void eat();
  void think();
  int hungry_p();
  void dining();
  void stopMe();
};

void Phil :: start() {
  Agent::start();
  meal_port = new BB_Port<int>(this,nV("MealBB"),INF);
  right_port
            = new BB_Port<int>(this,nV(int2intstr(myRight)),INF);
  left_port = new BB_Port<int>(this,nV(int2intstr(myLeft)),INF);
  if (myRight) {                  // If I'm not the first agent
    meal_port->put("Mealticket", aTicket);
    aTicket = 0;
  }
  left_port->put("fork", 1);    // Put my fork on the left side
  dining();                       // Enter dining phase
  stopMe();                       // Terminate
}
void Phil :: dining() {
  while (hungry_p()) {
    getMealticket(); getForks();
    eat();
    putForks(); putMealticket();
    think();
  }
}
void Phil :: getMealticket() {
  aTicket = meal_port->get("Mealticket");
}
void Phil :: getForks() {
  lFork = left_port->get("fork");
  rFork = right_port->get("fork");
}
void Phil :: putForks() {
  right_port->put("fork", rFork);
  left_port->put("fork", lFork);
  rFork = 0;
  lFork = 0;
}
void Phil :: putMealticket() {
  meal_port->put("Mealticket", aTicket);
  aTicket = 0;
}
void Phil :: stopMe() {
  meal_port->put("money", 1); // Pay
  Agent :: stopMe();          // And definitelly call termination
}
```

Fig. 6.16. The `Phil` class declaration and the implementation of its `start`, `dining`, `getMealticket`, `getForks`, `putForks`, `putMealticket`, and `stopMe` methods for the problem of the Dining Philosophers.

6.8 Implementation of a Prototype

6.8.1 Introduction

An implementation of a prototype of STL++ has been realized. It is a prototype in the sense that it has not been tested by a large set of users and applications, and that the implementation should be enhanced in order to improve the performances. This section sketches out the main characteristics of the realization.

The choice of the underlying platform was very important in the design of the implementation. Because our prototype was aimed at supporting the implementation of MASs on distributed architectures, a cluster of networked workstations has been opted as a hardware architecture. The ease of finding and taking profit from such kinds of resources was an additional reason. Concerning the underlying software architecture, we needed a communication platform allowing asynchronous communication over a cluster of workstations and a support for multithreading. Multithreading has the strong advantage of offering efficient communication for tightly-coupled threads within a same address space. Both characteristics are precisely key elements of the communication library Pt-pvm [KHS96], [Kro97] that has been developed at the PAI Group of the University of Fribourg. Pt-pvm has also been used to implement STL, thus showing that such an implementation is possible. For these reasons and in a desire of continuity in the research, our choice was naturally motivated for Pt-pvm. Concerning the target object-oriented language, our choice was C++ [Str97] because it allows an easy integration of Pt-pvm thanks its C interface and it produces efficient code.

Section 6.8.2 discusses the use of Pt-pvm. Our integration of concurrency and object-orientation is explained in section 6.8.3. The core implementation is presented from sections 6.8.4 to 6.8.8.

6.8.2 Pt-pvm

Pt-pvm, which is based on PVM [Sun90], [GBJ+94] and the POSIX thread library [Pos95] (see figure 6.17), is a library providing message passing and process management facilities at thread and process level for a cluster of workstations.

Pt-pvm possesses three basic abstractions (see figure 6.18):

- A *parallel virtual machine* offered by PVM, which includes several CPUs.
- One or more *process environments* (PE) that run on each CPU; a PE represents a heavy-weight process.
- One or more Pt-pvm *threads* that run on each PE. As threads are denoted with an id TID, they can be identified by a triple (CPU, PE, TID).

A Pt-pvm application is started with a specific number of CPUs and PEs running on these CPUs. As soon as a function has been declared as a

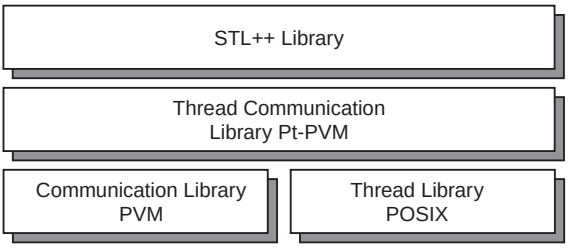

Fig. 6.17. Overview of the implementation layers of STL++.

thread, it can be spawn in three manners: (i) it can be explicitly launched on the same PE; (ii) it can be explicitly spawn on a specific CPU, where a PE in that CPU is chosen in a round-robin fashion; (iii) it can be spawn transparently on the network, letting the system choose the PE and CPU. Each thread runs in a *thread environment*, which contains information about its father, its identity and local parameters.

Pt-pvm has a mixed approach: both light-weight and heavy-weight processes can be started at runtime. Processes of both categories can also communicate with one another through the Pt-pvm communication primitives.

In order to communicate, Pt-pvm provides message-passing facilities similar to PVM. Processes are identified using a *Correspondent*, which is a tuple (PE, TID). Synchronous and asynchronous modes are supported.

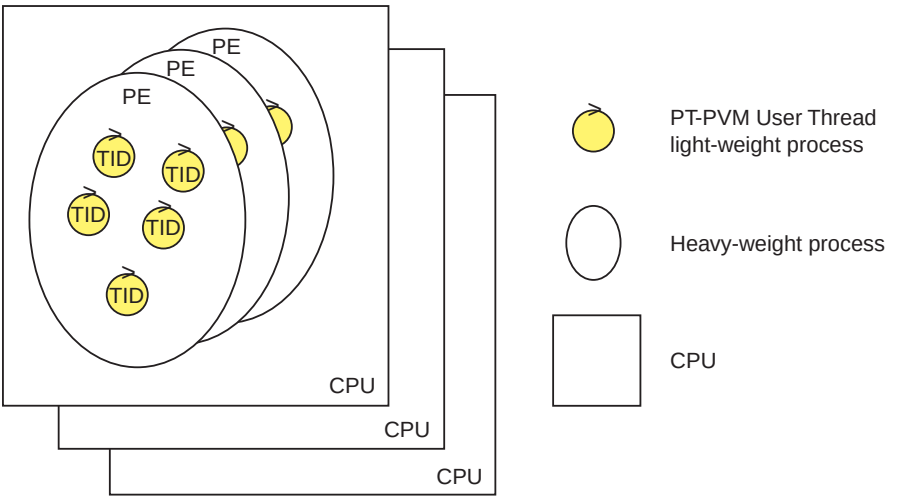

Fig. 6.18. Pt-pvm programming abstractions.

6.8.3 Concurrency and Object-Orientation Integration

One of the key design issues in the implementation of STL++ is the integration of concurrency and object-orientation, supporting simple *active objects* that have their own activity. Concurrent object-orientation is not only offered by STL++, but it is also used for implementing the different mechanisms underlying the implementation, like for supporting the matching of ports.

As there exist many approaches to concurrent object-orientation (for a review see for instance or [BGL98]), we precise our view. As both passive and active objects are available, STL++ has an heterogeneous approach. No internal object concurrency is supported: our active objects have a unique thread of execution. These characteristics are true for STL++ and the underlying active objects implementing STL++. Concerning request scheduling, an explicit acceptance of messages is realized: by reading explicitly on ports (case of STL++) or by receiving explicitly messages through PT-PVM primitives.

As PT-PVM creates new threads or processes by spawning functions with a specific profile, a simple scheme to render objects active has been chosen. Each active object class in the implementation inherits from `ThreadObj`, which supports basic facilities for thread activation like the PT-PVM correspondent and thread environment. Two pure virtual methods must then be re-implemented: `start` and `stopMe`. An appropriate spawning function (say f) is defined for launching active objects of a class c: it creates an instance o of c and calls its `start` method, which implements the acting of the object. Therefore, in order to create an active object, one has to use the spawning PT-PVM primitive `csBaseSpawn` that launches the function f. An application using STL++ calls the macros `defineBlop` and `defineAgent` on classes respectively inheriting from `Blop` and `Agent`. The purpose of these macros is precisely to define spawning functions f for blops and agents. The creation macros `createBlop` and `createAgent` call `csBaseSpawn` on spawning functions. Note that, in the underlying implementation, active objects always communicate with one another using PT-PVM facilities.

6.8.4 Blops and Agents

The `Blop` class inherits from `ThreadObj`. Using PT-PVM's `csBaseSpawn`, a wrapping spawning function of a blop is launched as a light-weight process transparently over the cluster within one PE. Thus, for the time being, a user does not have to indicate a CPU on which to launch the new process: it is chosen nondeterministically in the pool available. A blop locally creates two other active objects as threads: a `PortManager` object (see section 6.8.5) that is in charge of managing every blop port, and a `Matcher` that implements the matching of ports (see section 6.8.6) and setup connections (see section 6.8.7).

The `WorldBlop` class is similar to `Blop` with the difference that it does not define a `PortManager` object, because it cannot have ports.

The `agent` class also inherits from `ThreadObj`. However, in opposite to a blop, a wrapping spawning function of an agent is launched as a light-weight process and always locally, i.e. in the same PE as the invoker. An active `PortManager` object (see 6.8.5) is also created locally in order to manage the agent ports.

6.8.5 Ports and Port Managers

Port classes are implemented as passive template classes. For each entity (blop or agent), a `PortManager` active object is in charge of regulating the communication of the corresponding ports. The whole communication between an entity (i.e. also its ports) and its port-manager is realized through PT-PVM message-passing.

At its creation, a port announces itself to the port-manager, which in turn creates a proxy port (of class `PM_Port`) representing the announced port. When a message is posted on a port, it is automatically forwarded to the port-manager. This last one further forwards a copy of the message to all existing connections, or, if a port is unbound, stores the message in the FIFO buffer of the proxy port, which will be emptied by sending every message to a future new connection. By requiring a data item, a port sends a require message to the port-manager and blocks on a PT-PVM receive primitive. The port-manager checks if a message exist. If this is the case, it sends the message back to the concerned port; otherwise, the port-manager stores the demand and waits for another incoming PT-PVM message (e.g. new connection). When finally a value is available, it is sent back to the originally asking entity which will be unblocked.

When a port is deleted, its destructor method sends a corresponding message to the port-manager, so that the latter can inform the matcher, and handle concerned connections and remaining messages.

The management of blop ports has a speciality in its implementation, because the port-manager creates a pair of proxy ports for each blop port. This is due to the gateway role of such ports. Actually, a blop port has an internal view and an external view. Each proxy port of the pair is announced correspondingly to the matchers of the actual blop and of the father blop of the actual blop.

6.8.6 Matching

The matching between the ports of a same blop can be realized in two generic ways. In both cases, the needed messages are only exchanged between the entities of the concerned blop. In a *decentralized matching*, the testing of the matching and the decision to match are done for a port by its port-manager.

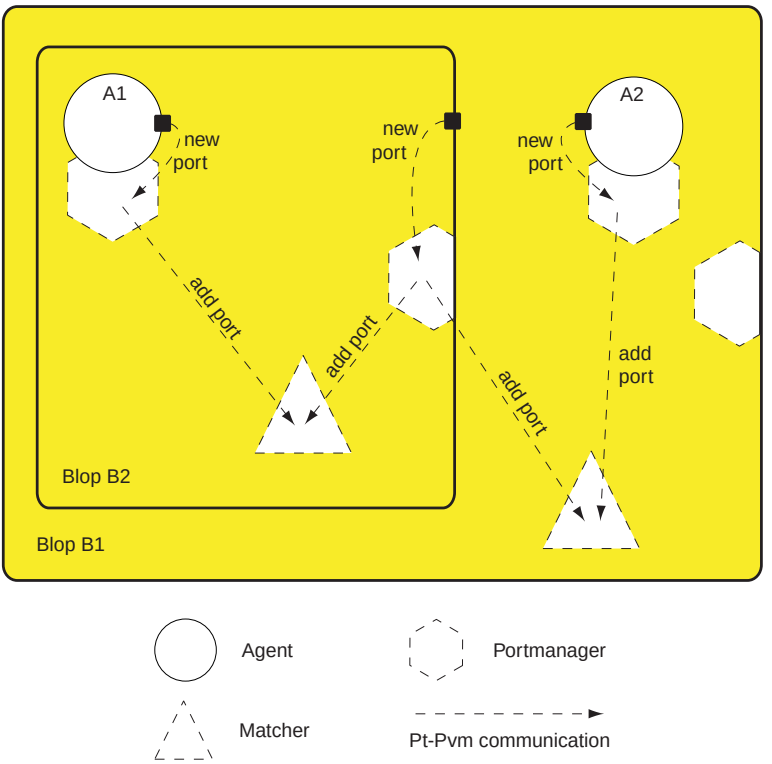

Fig. 6.19. Implementation of the matching mechanism of STL++.

This can be done by using a broadcast of the port signature to every other port-manager of the blop and applying a resultant protocol for deciding the participating ports in a connection.

Decentralized matching has the disadvantage of the high number of necessary messages (complexity of $O(n^2)$ for n port-managers). A *centralized matching* has been therefore adopted in our implementation, using a so-called *matcher* for each blop. A matcher is an underlying active object that has the task of checking possible matching compatibilities between port signatures in a specific blop. As explained above in section 6.8.5, a port-manager has received an appropriate message announcing the creation of a new port. The port-manager has sent to the matcher the announcement of this new port, asking to register it in its structure (defining a new proxy port of type `M_Port`) and test for possible matching. The matcher searches first for announced ports with an identical name and tests then the compliance between ports by applying the compatibility functions described in table 6.8 of section 6.5. With this scheme, decentralized matching has a message complexity of $O(n)$.

The whole mechanisms for the matching of STL++ are sketched in figure 6.19.

6.8.7 Connection Setup

Once a matcher has identified compliant ports, it has to setup a corresponding connection and send to the concerned port-managers useful information. Three generic schemes are possible.

For blackboard connections, the matcher launches an active object of class `BBC`, which implements a blackboard. It then sends to each concerned port-manager the PT-PVM correspondent of the blackboard. The blackboard will store data and reply to port-managers' requests.

Installing group connections has a similar scheme. An active object of class `GroupC` is started and its correspondent sent to the participating port-managers. Furthermore, the matcher indicates to the group connection the PT-PVM correspondents of each concerned port-manager. A group connection receives then messages of members of the group and automatically forwards them to every participant.

For streams, a similar approach as for the blackboard and group connections could have been chosen, by charging a new active object of storing and forwarding data to requesting port-managers. However, a different scheme has been adapted for efficiency concerns: messages are sent directly to the port-managers of the consumers and buffers are handled on that side. This way reduces importantly the number of PT-PVM messages but has the disadvantage of the implementation complexity in what concerns the handling of buffers. This is especially true when a port is destroyed and the data buffered in the resulting freed connection has to remain at disposal, so that new ports matching to a free connection really receive bufferized data.

6.8.8 Events

The implementation of events follows a simple scheme. The constructor of the class `Event` requires a reference on a `Condition` object. Each condition object implements the pure virtual method `test`. This method is called each time a `get`, `read` or `put` access is done to the port, or, for the `SaturatedCond` condition, when a connection is established. Launching an event simply calls the re-implemented `launch` method.

6.9 Discussion

In this chapter, we have presented STL++ as an instance of the ECM coordination model oriented towards the implementation of MASs. We discuss in the following paragraphs STL++ as a coordination language and its use for MASs.

6.9.1 STL++ as a Coordination Language

ECM along with STL++ its instantiation show similarities with several coordination models of the data-driven and process-oriented families. In this section, we accentuate general differences between ECM/STL++ and the main representatives of each class of coordination models and languages.

Though ECM and STL++ share many characteristics with IWIM [Arb96] and MANIFOLD [AHS93], they differ in several points:

- Blops are not coordinators like IWIM managers. Connections are not established explicitly by the blop (or by another agent), i.e. there is no language construct to bind ports in order to establish a connection: the establishment of connections is done implicitly, resulting from a matching mechanism between compatible ports within the same blop; the matching depends on the types and the states of the concerned ports. The main characteristics of blops is to encapsulate objects, thus forming a separate matching namespace for enclosed entities and an encapsulation mechanism for events.
- ECM/STL++ generalizes connection types, namely stream, blackboard or group, and does not restrict to channels. This is the reason why we consider ECM as a hybrid model.
- In ECM/STL++, events are not signals broadcast in the environment, but objects that realize an action in a blop. Events are attached to ports with conditions on their state that determine when they are to be launched. They can create agents or blops, and their action area is limited to a blop.

ECM together with STL++ differ in the following points from LINDA [CG89]:

- Like several further developments of the LINDA model, ECM/STL++ uses hierarchical multiple spaces [Gel89], in contrast to the single flat tuple space of the original LINDA. But it has to be stressed that a blop is not a shared dataspace as a tuple space, but an abstraction and encapsulation mechanism of several agents that serves as a separate name space.
- In ECM/STL++ agents get started either through an event in a blop, or automatically upon initialization of a blop, or through a creation operation by another agent; LINDA uses a single mechanism: `eval()`. The termination of an agent results in the loss of all its enclosed data and ports; it may as well result in the loss of pending data in the case of unbound connections. In LINDA, a tuple replaces the terminated process.
- In ECM/STL++ agents do not run in a medium which is used to transfer data. Every communication is realized through connections established by the matching of ports. Furthermore, these connections are private means: only the engaged agents have access to them.
- Blackboard connections allow for generative communication between the participating agents. For the time being, a string is used to introduce and

retrieve data. Neither tuples, nor templates, nor pattern matching are supported for retrieving data. Furthermore, blackboards are typed in the sense that only one type of data can be stored. There is also no kind of active tuples.

6.9.2 STL++ for MASs

Some characteristics that make STL++ adequate for implementing objective coordination in MASs are the following:

- Blops enable a grouping mechanism and the construction of a structured set of different environments, each of which is a closed private coordination space with communication interfaces to other environments.
- Agents are implemented as black boxes. In this way, a basic operational autonomy is ensured, because an agent owns exclusive control over its internal state and behavior (it acts pro-actively), and it can define by itself its ports. Furthermore, the internal structure of an agent is not accessible by other entities: an agent is seen from external views as a delineated entity presenting clear interfaces.
- Sensing and acting capacities of agents are implemented with ports. Thus agents can perceive and act through different means by virtue of the possibility of creating several instantiations of port types. A port perceives specific data: this is enforced with the data type feature of ports.
  As communication is only realized anonymously, we comply more with our target, because agents only act according to what they have sensed through their ports.
- Peer-to-peer, group and blackboard communication means allow to cover basic agent communication needs. Furthermore, ensuring a matching mechanism that is not centralized by a coordinator agent respects more the autonomy of the agents and the decentralized nature of MASs.
- As agents and environments may evolve over time, dynamicity is an important point in MASs. This has led to design STL++ so as to support dynamic blop and agent creation and termination, blop reorganization, new port creation and destruction, thus yielding to dynamic communication topologies.

STL++, however, has two important disadvantages. Firstly, in order to let agents migrate from one blop to the other, one has to integrate in its application intra-blops connections and adequate specialized agent creators; this is not necessarily a natural way to do it. Indeed, it would be very advantageous to give explicitly to the agents the ability to move from one blop to other. Secondly, our implementation is badly portable.

These two considerations motivated us to start the design and development of a new ECM instance, namely Agent&Co. This new coordination

language, which is still an ongoing project, slightly extends ECM in order to support mobility. Furthermore, the language is implemented in JAVA [GJS96] over an object request broker architecture [OMG97b] in order to enhance portability. AGENT&CO is the subject of the next chapter.

7. The AGENT&CO Coordination Language

7.1 Introduction

The main motivations for the design of a new ECM instance was the desire of studying agent mobility in the context of ECM and experimenting an integration of an ECM-grounded coordination language in an object request broker architecture [OMG97b] in order to enhance portability. For that aim, a new coordination language named AGENT&CO has been designed; it has been implemented as a library [Doi99], like STL++.

This chapter gives a general description of this new language binding, trying to show specific characteristics differentiating AGENT&CO from STL++. Note that this project is still underway, and has therefore to be enhanced from the experience that will be acquired with testing concrete applications. For these reasons, we review AGENT&CO on a general ground, skipping details.

Sections 7.2 to 7.6 review the instantiations of each ECM construct, sticking to the essential. Section 7.7 discusses choices in a realized implementation.

7.2 Blops

As the general ECM semantics asks for, an AGENT&CO blop has the roles of encapsulation of other objects and of a separate name space, allowing the matching of ports and the creation of connections inside a blop.

But how can we provide agents with migration capacity? Mobility offers indeed agents the ability of moving from one space to the other. The chosen way for supporting mobility is then directly dependent from the semantics of the underlying logical localities, like the use of access rights for accessing contained data or the identification of such localities. The integration of mobility for ECM can therefore at least be done in two ways. In both cases, the unit of locality is naturally the blop.

A first way, more respectful of the whole philosophy of ECM, would give the agents the ability to move themselves by calling a move operation on a specific own port. This move operation would post the actual agent state on the concerned port. A specialized creation agent would then receive this state through a connection (in the same or in another blop) and recreate the arrived agent. Such a scheme has been implemented in STL++

M. Schumacher: Objective Coordination in MAS Engineering, LNAI 2039, pp. 97–101, 2001.

for a simulation of mobile collective robotics (see chapter 8). This solution, however, has two disadvantages. Firstly, it does not really integrate mobility in the coordination language, because specialized agents have to be defined. Secondly, it introduces an uncontrolled cloning mechanism in case an agent port is bound to several connections. For those reasons another mobility integration scheme has been adopted.

Indeed, the second integration possibility introduces a naming mechanism for blops. If an agent wants to move to another blop, it calls an own move method towards another identified blop, where the agent will migrate and continue its activity. Section 7.3 explains this mechanism in more details. It is important to see that only blops are named; agents obviously remain unidentifiable.

The identification of blops is done using a mechanism similar to file names, by indicating absolutely or relatively the hierarchy of blops. For instance, taking the configuration of figure 7.2, one can identify a blop by `"/B1/B3"`. Note that blop identifications can also be exchanged between agents through connections.

Another speciality of AGENT&CO is the creation of a *blop blackboard* for each blop. This blackboard, which is untyped, is by default available for each agent inside the blop. Furthermore, the introduction of a blop identification mechanism allows external agents to post data into another blop blackboard by indicating the name of the respective blackboard's blop.

7.3 Agents

Like in STL++, ECM processes in AGENT&CO are named agents, because they are intended at supporting their implementation. They can be created by blops, other agents, and events.

In AGENT&CO, we define a novelty concerning message management. A FIFO *arrival list* for KK-Streams and group connections is introduced in order to keep track of the order in which messages have arrived. Each time a message has arrived in such a connection, a reference to the respective connected port is introduced in this FIFO list. An agent has thus the possibility to consult this list in order to decide which port will be accessed. When a message has been read, the reference to the port is removed from the list. But this list always remains an optional way to read data on ports.

If we permit agents to move from one blop to the other, we must ask ourselves under which conditions an agent can migrate, and how this is realized? At any moment, any agent port may be bound to connections. Thus, an agent has to disconnect itself from any connection before it migrates from one blop to the other. If this condition is fulfilled, an agent can call a `move` method indicating its target blop, where it is re-initialized and restarted. For the re-initialization, a `reinit` method is called. This method, which has been re-implemented by the agent class, typically re-creates new ports.

7.4 Ports and Connections

AGENT&CO defines four port features. The first two features define the basic port classes. The other features are used as arguments in the ports constructors. Supported port features encompass:

- *Communication structure*, which captures the basic ECM communication paradigms;
- *Orientation*, defining the direction of the data flow;
- *Saturation*, defining the number of connections a port can have;
- *Data type*, defining the type of data authorized to pass through a port.

The basic types of ports encompass the following:

- Unidirectional *KK-Stream* ports; the connection resulting from their connection results in a KK-Stream.
- *Blackboard* ports. The resulting connection has a similar semantics to the semantics of the STL++'s blackboards. The connection is persistent when all corresponding ports disappear.
- *Group* ports, with the usual closed group semantics.

Every communication is understood to be asynchronous. Thus, no synchronous streams are supported.

At their creation, ports are assigned names, which are only used for the matching mechanism. Blop ports are particular in this respect, because there exist two types of names: some are available for the matching at the outside of the blop, and others at the inside.

7.5 Matching

The matching rules follow directly the basic matching conditions of ECM, with the particularity of double names for blop ports (see above). Table 7.1 indicates the compatibility functions between the features.

Table 7.1. Compatibility for features F for two ports P1 and P2.

Feature F	**Values**	**Compatibility**
Communication	`blackboard`, `stream`, `group`	`P1.F` = `P2.F`
Orientation	`in, out, inout`	(`P1.F` = `in` *and* `P2.F` = `out`) *or* (`P1.F` = `P2.F` = `inout`)
Saturation	`{1,2,...,INF}`	Always compatible
Data Type	Type	`P1.F` = `P2.F`

7.6 Events

Events are similar to STL++ events. A launch method defines the acting of the event. Furthermore, an event is created with a specific lifetime, which is decremented each time the event is launched. For each existing condition, a port class has a specific method that attaches an event to that port. Supported conditions encompass the STL++ *AccessedCond*, *PutAccessedCond*, *GetAccessedCond* and *UnboundCond*.

7.7 Implementation

A prototype implementation of AGENT&CO has been realized as a framework of JAVA classes [GJS96]. It is based on the ORBACUS object request broker [OOC99]. In this section, two design issues of the implementation are briefly discussed.

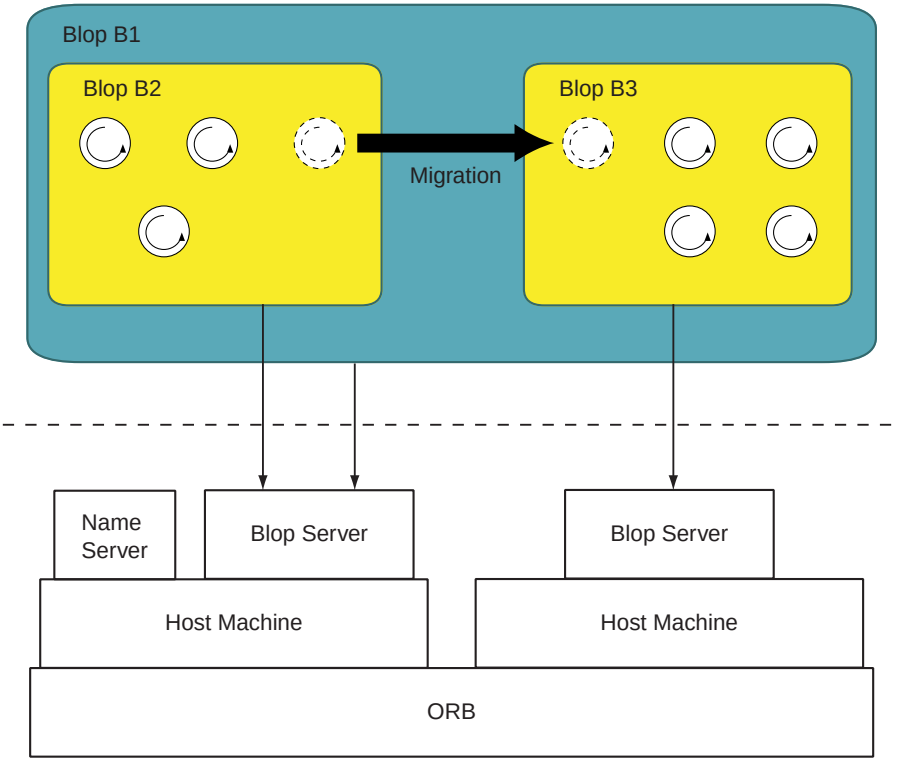

Fig. 7.2. The ORB integration of AGENT&CO.

Blops are hosted on so-called *blop servers*. A set of blop servers is started over the physical nodes and communicate through the ORB. A *name server* is also launched in order to support the logical naming structure of the blops. Actually, the identification tree is realized thanks to the CORBA *CosNaming* service [OMG97a]. As each blop is related to a specific naming context, a path for a blop is obtained by adding the naming context of this blop to its parent's naming context. Figure 7.2 sketches these design issues.

AGENT&CO agents are implemented as JAVA threads. The support of the migration mechanism uses a specific scheme. Before to move, an agent object is first translated into an array of bytes and then received by the target blop, which will restore the arrived object. The translation into an array of bytes uses JAVA's serialization. It is the task of the source blop to serialize the migrating object. At reception, the new agent is actualized to its new context and re-initialized; a new thread is then created and started.

Further details on the implementation can be found in [Doi99].

8. Collective Robotics Simulation

8.1 Introduction

In section 2.5, our specific class of MASs has been introduced. We have also described a generic model belonging to this class (section 2.5.1) and presented a peculiar application of this model (section 2.5.2). In this chapter, the design and the implementation of this application with ECM/STL++ is presented. Figure 8.12, at the end of this chapter, summarizes the whole process from the model to the implementation.

In the *Environment*, which is a torus grid, every cell has four neighbors. By applying a four connectivity, the implementation is facilitated, and this does not fundamentally vary the functionality from a more complex scheme such as an eight connectivity. The agents that roam on this environment are named SimRobots, because they simulate the robots. They comply the agent model introduced in figure 2.6 of section 2.5: the agents sense the environment through their sensors and act upon their perception at once.

To take advantage of distributed systems, the *Environment* is split into N sub-environments, each being encapsulated in a blop, as depicted in figure 8.1, thus providing an independent functioning between sub-environments (and hence between agents roaming in different sub-environments). Note that blops have to be arranged so as to preserve the topology of the sub-environments they implement.

8.2 Global Structure

The world blop `RoboticSimulationWorld`, which is the world blop, is composed of an `init` agent, in charge of the global initialization of the system, and a set of `N` pre-defined blops, each encapsulating and handling a sub-environment. On figure 8.2 can be found the declaration of the world class and the implementation of its constructor.

Figure 8.11, at the end of this chapter, gives a graphical overview of the organization within a sub-environment blop in the world blop. Note that on this figure, only one sub-environment blop is represented in order to avoid cluttering the picture. In the case of an application with multiple sub-environment

M. Schumacher: Objective Coordination in MAS Engineering, LNAI 2039, pp. 105–114, 2001.

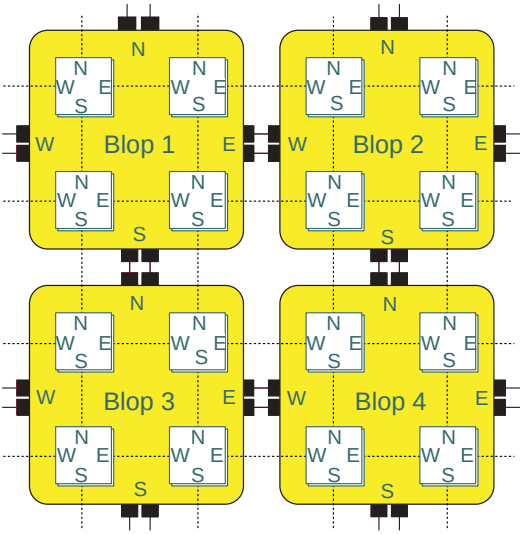

Fig. 8.1. Environment split up among 4 blops.

blops, there should be some connections between the `init` agent and all the blops, as well as some connections between north ports of top blops with south ports of bottom blops and east ports of east blops with west ports of west blops; they have been intentionally dropped.

Figure 8.3 shows the class hierarchy of the implementation. In the next sections, we explain one by one the implementation classes.

```
1  class RoboticSimulationWorld : public WorldBlop {
2  public:
3    RoboticSimulationWorld();
4    ~RoboticSimulationWorld();
5  };
6  RoboticSimulationWorld::RoboticSimulationWorld() : WorldBlop() {
7    int sx = SIZE_X, sy = SIZE_Y;
8    int se = NBE_SUBENV, no = NBE_OBJECTS, na = NBE_AGENTS;
9    for(int i=0; i<se; i++) {
10     // create subenv with basic topologies (n, s, w, e)
11     createBlop(SubEnv_B, &se,
12                &topo[i,0], &topo[i,1], &topo[i,2], &topo[i,3]);
13   }
14   createAgent(init, &se, &sx, &sy, &no, &na);
15   stopMe();
16 }
```

Fig. 8.2. The `RoboticSimulationWorld` world class declaration and the implementation of its constructor.

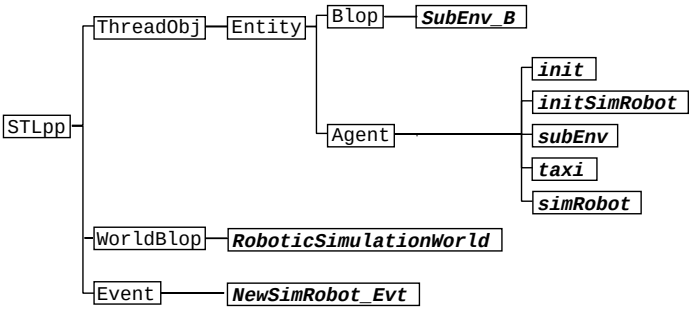

Fig. 8.3. Robotic Simulation class hierarchy.

8.3 init Agent

The init agent (see class definition in figurels 8.4) has four static ports for every blop to be initialized: three of type KK_outPort (init_Nb_Agts_P, cre_SimRbs_P and cre_SubEnv_P) and one of type KK_inPort (eot_P).

The role of the init agent is threefold: i) create through its init_Nb_Agts_P and cre_SimRbs_P ports the initial agents within every blop; ii) set up through its cre_SubEnv_P port the sub-environment (size, number of objects and position of the objects on the cells) of every blop; iii) collect the result of an experiment, signal the end of an experiment, and properly shutdown the system through the eot_P port.

```
1  class init : public Agent {
2   public:
3    init(int, int, int, int, int);
4    ~init();
5    void start();
6   private:
7    int size_x_env, size_y_env;
8    int nbr_objects, nbr_agents;
9    KK_outPort<InitInitAgent> *init_Nb_Agts_P;
10   KK_outPort<RbState> *cre_SimRbs_P;
11   KK_outPort<InitSubEnv> *cre_SubEnv_P;
12   KK_inPort<int> *eot_P;
13 };
```

Fig. 8.4. The init class declaration.

8.4 Sub-environment Blops

Figure 8.5 shows the declaration of a sub-environment class (SubEnv_B) and the implementation of its start method, both in conformance with the application depicted in figure 8.11.

```
class SubEnv_B : public Blop {
 public:
  SubEnv_B(int, int, int, int, int);
  ~SubEnv_B();
  void start();
 private:
  char* se_pos, north, south, west, east;  // For port names
  KK_Blop_inPort<RbState> *i_SimRobots_P;
  KK_Blop_inPort<InitSubEnv> *i_SubEnv_P;
  KK_Blop_inPort<InitInitAgent> *i_NbrAgts_P;
  KK_Blop_outPort<int> *eot_P;
  KK_Blop_inPort<RbState> *north_i_P;
  KK_Blop_outPort<RbState> *north_o_P;
  KK_Blop_inPort<RbState> *south_i_P;
  KK_Blop_outPort<RbState> *south_o_P;
  KK_Blop_inPort<RbState> *west_i_P;
  KK_Blop_outPort<RbState> *west_o_P;
  KK_Blop_inPort<RbState> *east_i_P;
  KK_Blop_outPort<RbState> *east_o_P;
};

void SubEnv_B::start() {
  Blop::start();

  i_SimRobots_P =
   new KK_Blop_inPort<RbState>(this,
                               nV(catStr("INIT_AGENTS",se_pos)),
                               1,INF);
  i_SubEnv_P =
   new KK_Blop_inPort<InitSubEnv>(this,
                               nV(catStr("INIT_SUBENV",se_pos)),
                               1,INF);
  i_NbrAgts_P =
   new KK_Blop_inPort<InitInitAgent>(this,
                               nV(catStr("NB_AGTS",se_pos)),
                               1,INF);
  eot_P =
   new KK_Blop_outPort<int>(this,
                            nV(catStr("END_OF_TASK",se_pos)),
                            1, INF);
  north_i_P = new KK_Blop_inPort<RbState>(this,nV(north),1,INF);
  north_o_P = new KK_Blop_outPort<RbState>(this,nV(north),1,INF);
  south_i_P = new KK_Blop_inPort<RbState>(this,nV(south),1,INF);
  south_o_P = new KK_Blop_outPort<RbState>(this,nV(south),1,INF);
  west_i_P = new KK_Blop_inPort<RbState>(this,nV(west),1,INF);
  west_o_P = new KK_Blop_outPort<RbState>(this,nV(west),1,INF);
  east_i_P = new KK_Blop_inPort<RbState>(this,nV(east),1,INF);
  east_o_P = new KK_Blop_outPort<RbState>(this,nV(east),1,INF);

  createAgent(initSimRobot);
  createAgent(subEnv);
  createAgent(taxi);
}
```

Fig. 8.5. The `SubEnv` class declaration and the implementation of its `start` method.

In a sub-environment, several agents are created. They can be distributed in two categories: agents that are purpose-built for a distributed implementation of the multi-agent application (they enable a distributed implementation), namely `initAgent` and `taxi`, and agents that are actually peculiar to the multi-agent application, viz. `subEnv` and `simRobot` agents.

Each sub-environment blop has twelve ports: four *KK-Stream out-flowing direction* ports (`north_o_P`, `south_o_P`, `west_o_P`, and `east_o_P`) and four *KK-Stream in-flowing direction* ports (`north_i_P`, `south_i_P`, `west_i_P`, and `east_i_P`), which are all gateway ports enabling agent migration across blops; three *in-flowing KK-Stream* ports, namely `i_NbrAgts_P`, `i_SimRobots_P` and `i_SubEnv_P` used for the creation of the initial `SimRobot` agents (actually realized in the `initSimRobot` agent) and for an appropriate setup of the sub-environment (achieved in the `subEnv` agent); and an *out-flowing KK-Stream* port (`eot_P`) used to forward to the `init` agent the result of an experiment and to indicate the end of an experiment.

8.5 initSimRobotAgent, NewSimRobot_Evt Event

The `initSimRobot` agent (see figure 8.6) is responsible for the creation of the `SimRobot` agents. It has five ports: `nb_Agts_P` (*in-flowing KK-Stream port*), `newArrival_P` (*in-flowing KK-Stream port*), `location_o_P` and `location_i_P` (*out-flowing* and *in-flowing KK-Stream ports*), and `init_P` (*out-flowing KK-Stream port*).

```
1  class initSimRobot : public Agent {
2   public:
3    initSimRobot();
4    ~initSimRobot();
5    void start();
6   private:
7    KK_inPort<RbState> *newArrival_P;
8    KK_outPort<RbState> *init_P;
9    KK_inPort<InitInitAgent> *nb_Agts_P;
10   KK_outPort<int> *location_o_P;
11   KK_inPort<int> *location_i_P;
12 };
```

Fig. 8.6. The `initSimRobot` class declaration.

At the outset of the experiment, the `initSimRobot` agent through its `nb_Agts_P` port will be informed by the `init` agent on the number of agents to be created in the present blop. The `initSimRobot` agent will then loop on its `newArrival_P` port so as to receive starting information of the agents to be created. As soon as a value comes to this port, the `NewSimRobot_Evt` event (see figure 8.7), which has been attached to `newArrival_P` with the `AccessCond` condition, is triggered and creates a new `SimRobot` agent. In the meantime, the `initSimRobot` agent draws randomly for every SimRobot

a starting position. The location_i_P and location_o_P ports enable the initSimRobot agent to communicate with the subEnv agent, so as to ensure that no more than one SimRobot agent can reside on an empty cell. The initSimRobot agent will then write on its init port some values for the SimRobot agent just created. The latter, through its creation_P port will read the information that was previously written on the init_P port of the initSimRobot agent. Values that are transmitted feature for instance the position of the SimRobot agent and its *state*.

```
1  class NewSimRobot_Evt : public Event {
2   public:
3    NewSimRobot_Evt();
4    ~NewSimRobot_Evt();
5    void launch();
6  };
7
8  NewSimRobot_Evt :: NewSimRobot_Evt()
9    : Event(new AccessedCond(), INF) {
10 }
11
12 void NewSimRobot_Evt :: launch() {
13   createAgent(simRobot);
14 }
```

Fig. 8.7. The NewSimRobot_Evt class declaration, and the implementation of its constructor and launch method.

Note that the newArrival_P port is connected to all *in-flowing direction* ports of the blop within which it resides, thus enabling to deal with migrating SimRobot agents across blops in the course of an experiment, by the same event mechanisms as described above. The location_i_P and location_o_P ports are very useful at this point, because the initAgent agent can already check with the subEnv agent whether the position to which the SimRobot agent intends to move is permitted or not. In case it is not, the initAgent agent will have to draw randomly a new position in the neighborhood of the position where the agent intended to go.

8.6 SimRobot Agent

This agent (see figure 8.8) has three ports: req_ans_P of type BB_Port, creation_P of type KK_inPort, and taxi_P a KK_outPort port. As already stated, this agent reads on its creation_P port some values (its position and its state). All req_ans_P ports of the agents are connected to a *blackboard*, through which agents sense their environment (*perception*) and act (*action*) into it, by performing *get/put* operations with appropriate messages. The type of action an agent can perform depends on the type of *control algorithm* implemented within the agent (see the architecture of an agent on figure 2.6). The taxi_P port is used to communicate dynamically with the taxi agent in

case of migration: the position and *state* of the agent are indeed copied to the `taxi` agent. The decision of migrating is always taken by the `subEnv` agent.

```
1  class simRobot : public Agent {
2   public:
3    simRobot();
4    ~simRobot();
5    void start();
6   private:
7    KK_inPort<RbState> *creation_P;
8    BB_Port<TemplateMsgFromAgent> *req_ans_P;
9    KK_outPort<RbState> *taxi_P;
10   int id;
11   int offset;
12   float alpha;
13   int carrying;
14   long rand_walk_seed;
15   long rand_pick_up_seed;
16   char myName[10];
17 };
```

Fig. 8.8. The `SimRobot` class declaration.

8.7 subEnv Agent

The `subEnv` agent (see figure 8.9) handles the access to the sub-environment and is in charge of keeping data consistency. It is also responsible for migrating `SimRobot` agents, which will cross the border of a sub-environment.

```
1  class subEnv : public Agent {
2   public:
3    subEnv();
4    ~subEnv();
5    void start();
6   private:
7    KK_inPort<InitSubEnv> *init_P;
8    KK_outPort<RbState> *to_taxi_P;
9    BB_Port<TemplateMsgFromAgent> *in_out_P;
10   KK_outPort<int> *location_o_P;
11   KK_inPort<int> *location_i_P;
12   KK_outPort<int> *eot_P;
13 };
```

Fig. 8.9. The `subEnv` class declaration.

It has an `in_out_P` port (of type `BB_Port`) connected to the *blackboard* and a `KK_outPort` port `to_taxi_P` connected to the `taxi` agent. Once initialized through its `init_P` `KK_inPort` port, the `subEnv` agent builds the sub-environment. By performing *get*/*put* operations with appropriate messages, the `subEnv` agent will process the requests of the `SimRobot` agents (e.g. number of objects on a given cell, move to next cell) and reply to their requests

(e.g. x objects on a given cell, move allowed and registered). When the move of a `SimRobot` agent leads to cross the border (cell located in another blop), the `subEnv` agent will first inform the `SimRobot` agent that it has to migrate, then inform the `taxi` agent that a `SimRobot` agent has to be migrated (the direction the agent has to take will be transmitted, so that the `taxi` agent can know which port to write to). The `location_i_P` and `location_o_P` ports are used to communicate to the `initAgent` agent further to its request on the position of an `SimRobot` agent with regard to other SimRobots' and objects' positions.

8.8 `taxi` Agent

The `taxi` agent (see figure 8.10) is responsible for migrating agents across blops. It has four static `direction` ports (of type `KK_outPort`), which are connected to the four *out-flowing direction* ports of the blop within which it stands. When this agent receives on its static `KK_inPort` port `requ_P` the direction towards where an agent has to migrate, it will dynamically create a `KK_inPort` port `con_SimRbs_P` in order to establish with the appropriate `SimRobot` agent a communication, by means of which it will collect all the useful information of the `SimRobot` agent (intended position plus *state*). These values will then be written on the port corresponding to the direction to take and will be transferred to the `newArrival_P` port of the `initAgent` agent of the concerned blop inducing the dynamic creation of a new `SimRobot` agent in the blop, thus materializing the migration.

```
1  class taxi : public Agent {
2   public:
3    taxi();
4    ~taxi();
5    void start();
6   private:
7    KK_inPort<RbState> *requ_P;
8    KK_inPort<RbState> *con_SimRbs_P;
9    KK_outPort<RbState> *tNorth_P;
10   KK_outPort<RbState> *tSouth_P;
11   KK_outPort<RbState> *tWest_P;
12   KK_outPort<RbState> *tEast_P;
13 };
```

Fig. 8.10. The `taxi` class declaration.

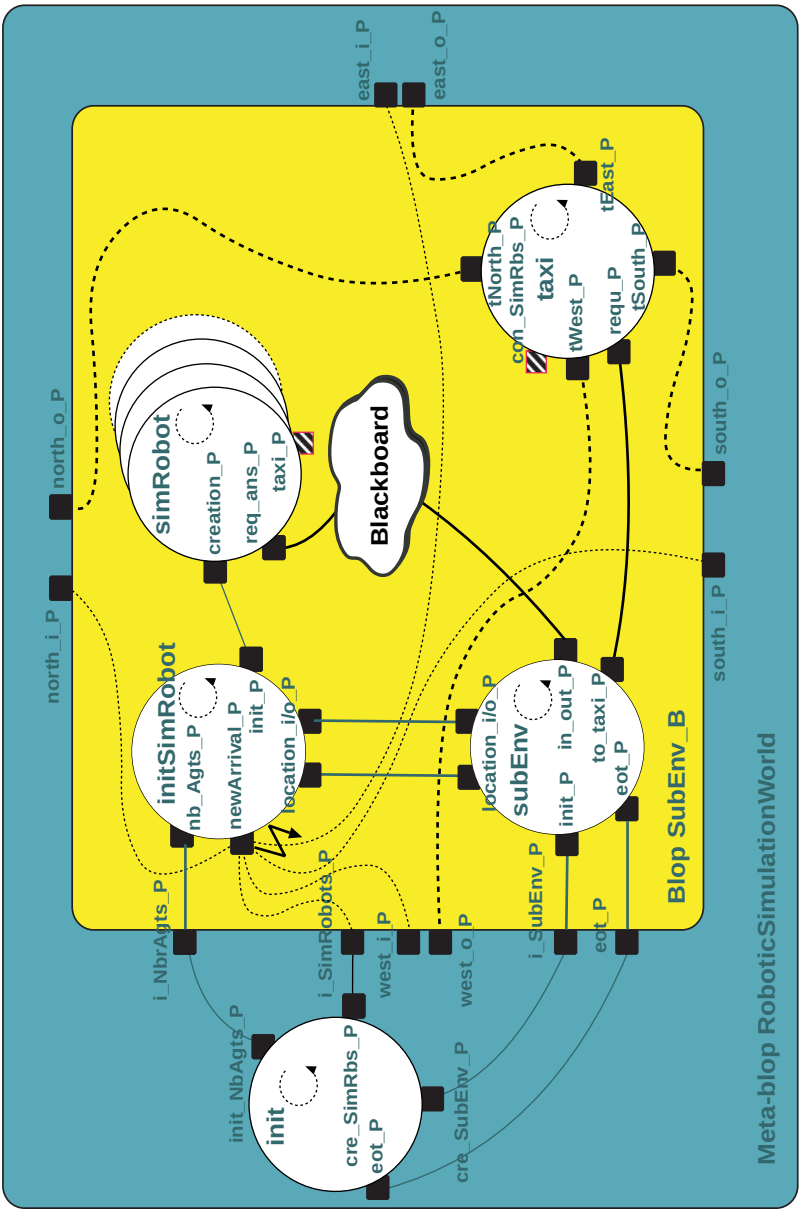

Fig. 8.11. `init` agent and a single sub-environment blop: solid and dotted lines are introduced just for a purpose of visualization.

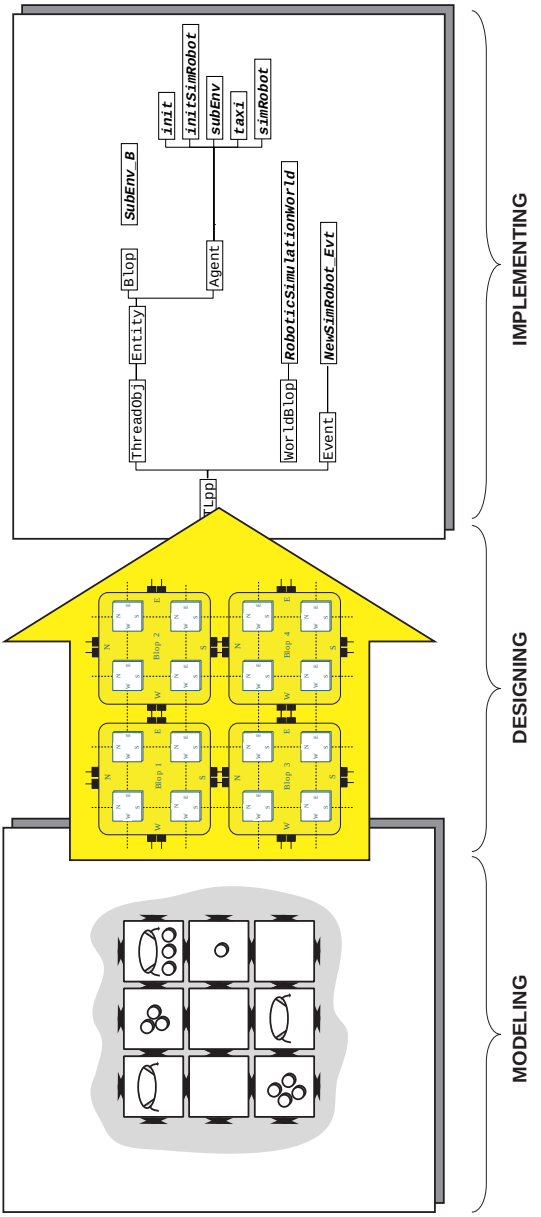

Fig. 8.12. Collective robotics simulation: from the model to the implementation.

9. Trading System Simulation

9.1 Introduction

STL++ has primarily been designed for supporting the implementation of MASs comprising simulated embodied agents. However, the coordination language may also be used to implement other classes of MASs. For that reason, we show in this chapter how STL++ can be used for implementing a simulation of a trading system [SCK+99], [SCH99].

Indeed, the numerous activities that take place within a trading system are typically distributed and can be modeled by a MAS. Agent-based electronic commerce is therefore becoming one of the most important application domains for MASs [NRS+98], [NS98], [MGM99]. Concrete solutions encompass several agent-based frameworks like KASBAH [CM96] or FISHMARKET [RANSP97].

The research in coordination models and languages has also proposed some specific scenarios for implementing electronic commerce applications; see for instance a proposal based on the PAGESPACE platform [CKR+97] and another one using IWIM and MANIFOLD [PA98b].

Our goal is not to achieve full automation of a trading system. This work rather concentrates on simulating the automation of such a system. Furthermore, our aim is neither to focus on the control algorithms of the different agents, nor on the negotiation techniques (see e.g. [GM98]) that are undertaken by the agents in order to process a transaction, but rather to concentrate on the basic coordination mechanisms (the objective coordination) that come into play in the interactions between agents, for which STL++ is precisely suitable. Thus, in this implementation, agents are endowed with a very basic autonomy in the sense that they can make decisions on their own, without the intervention of the user. More sophisticated autonomy-based control algorithms and smart negotiation techniques could be tackled in a further stage.

Figure 9.7, at the end of this chapter, gives a scaled down graphical overview of the organization of the agents that compose our trading system, as well as their interactions. To avoid cluttering the graph, port names (on which the matching is based) have been intentionally omitted. Figure 9.1 shows the class hierarchy of the implementation discussed in this chapter. We explain the different classes in the following sections.

M. Schumacher: Objective Coordination in MAS Engineering, LNAI 2039, pp. 115–120, 2001.

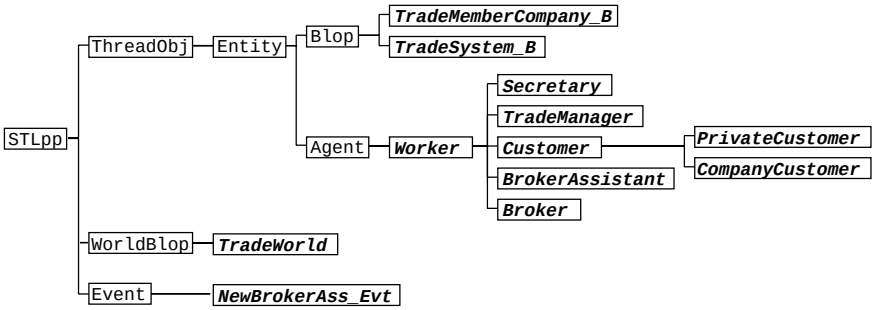

Fig. 9.1. Trading System class hierarchy.

9.2 The *TradeWorld* Blop

The `TradeWorld` blop (see figure 9.2) confines every activity in the trading simulation. Several *Trade System Blops* (TSB) are accessible by *customers* (company or private customers), who are authorized members of a trading system that represent end-user agents.

In our trading world, *Company* or *Private Customers* are either members of a `TradeMemberCompany_B` blop (`CompanyCustomer` agent), or act on their own (`PrivateCustomer` agent). They create queries to buy or sell goods. These queries, written by the customer on his `query_P` port (of `KK_outPort` type), are transmitted to a *Trading System Blop*.

```
1  class TradeWorld : public WorldBlop {
2  public:
3    TradeWorld();
4    ~TradeWorld();
5  };
6
7  TradeWorld :: TradeWorld() : WorldBlop() {
8    createBlop(TradeSystem_B);
9    createBlop(TradeMemberCompany_B);
10   createAgent(PrivateCustomer);
11   // ... further creations
12   stopMe();
13 }
```

Fig. 9.2. The `TradeWorld` class declaration and the implementation of its constructor.

9.3 The Brokers and the Broker Assistants

In a TSB (of type `TradeSystem_B`) reside `Broker` agents (see figure 9.3), each of whom is devoted to serve a particular customer, by handling customer queries committed to him. A pair of ports on the TSB, namely

*name*_query_gate_P and *name*_res_gate_P (*name* being `off1` or `off2` in figure 9.7), serves as gates for each customer and his respective broker. Every query is then posted by the broker to his `trade_P` port (of `BB_Port` type), e.g., sell 100 securities at 1000 CHF and therefore published in the *Trade Unit Blackboard*.

```
1  class Broker : public Worker {
2   public:
3    Broker(char*);
4    ~Broker();
5    void start();
6   private:
7    char* repr_name;
8    Query q;
9    TradeUnit tu;
10   NewBrokerAss_Evt *evt;
11   KK_inPort<Query> *query_P;
12   BB_Port<TradeUnit> *trade_P;
13   TradeUnit make_TradeUnit(Query*);
14 };
15
16 void Broker :: start() {
17   Agent::start();
18   query_P =
19     new KK_inPort<Query>(this, nV(catStr("query_", repr_name)),
20                          1, INF);
21   trade_P = new BB_Port<TradeUnit>(this, nV("trade"), INF);
22   while (continue_p()) {
23     q = query_P->get();
24     evt = new NewBrokerAss_Evt(repr_name, q.customer_id,
25                                q.transaction_id);
26     trade_P->attachEvent(evt);
27     tu = make_TradeUnit(&q);
28     trade_P->put("query", tu);
29   }
30   stopMe();
31 }
```

Fig. 9.3. The `Broker` class declaration and the implementation of its `start` method.

On the `trade_P` port is bound the event `NewBrokerAss_Evt` (see figure 9.4) with the condition `PutAccessedCond`; the effect of the posting is that the event `NewBrokerAss_Evt` is triggered. The role of this event is to dynamically create a `BrokerAssistant` agent that will be in charge of fulfilling the specific query, by means of a dynamic connection with another broker assistant interested in an (almost) symmetric query (e.g., buy 50 securities at 1000 CHF). This is possible on the basis of the transmitted information by the *Trade Manager*.

9.4 The Trade Manager

All queries are supervised by the `TradeManager` agent (see figure 9.5); he knows who issued which query (through an identification contained in the

```
class NewBrokerAss_Evt : public Event {
 public:
  NewBrokerAss_Evt(char*, Customer_id, int);
  ~NewBrokerAss_Evt();
  void launch();
 private:
  char* repr_name;
  Customer_id cust_id;
  int trans_no;
};

NewBrokerAss_Evt :: NewBrokerAss_Evt(char* name,
                                     Customer_id cid, int no)
  : Event(new AccessedCond(), 1) {
    repr_name = name;
    cust_id = cid;
    trans_no = no;
}

void NewBrokerAss_Evt :: launch() {
  createAgent(BrokerAssistant, &repr_name, &cust_id, &trans_no);
}
```

Fig. 9.4. The `NewBrokerAss_Evt` class declaration and the implementation of its constructor and its `launch` method.

query). For each new arrived query, the Trade Manager checks whether a possible matching between proposals can take place (e.g. case of a broker who wants to buy securities of type A and another broker who wants to sell securities of type A). If a kind of matching can somehow be issued between two *Broker Assistants*, the Trade Manager puts on the *Trade Unit Blackboard* two appropriate messages for each involved *Broker Assistant*. The information transmitted contains among others a specific transaction id.

```
class TradeManager : public Worker {
 public:
  TradeManager();
  ~TradeManager();
  void start();
 private:
  int counter;
  BB_Port<TradeUnit> *trade_P;
  TradeTable trade_table;
  bool try_to_match(TradeUnit*, TradeUnit*, TradeUnit*);
  bool trade(TradeUnit*, TradeUnitList*, TradeUnit*, TradeUnit*);
};
```

Fig. 9.5. The `TradeManager` class declaration.

9.5 Transactions

A newly created `BrokerAssistant` agent (see figure 9.6) has first to read from his `trade_P` port (of type `BB_Port`) in order to be informed of a specific transaction. In virtue of this information, he dynamically publishes two

KK_Ports using the transaction id as port name (trade_partner_i_P and trade_partner_o_P ports). A new double connection is then established between these two partners. Both involved Broker Assistants exchange useful information in order to make their query successful (this is precisely where negotiation techniques appear).

```
class BrokerAssistant : public Worker {
 public:
  BrokerAssistant(char*, Customer_id, int);
  ~BrokerAssistant();
  void start();
 private:
  char* repr_name;
  Customer_id customer_id;
  int transaction_id;
  Result answ;
  TradeUnit tu;
  KK_outPort<Result> *result_P;          // For transmitting results
  BB_Port<TradeUnit> *trade_P;           // Trade BB
  KK_outPort<int> *trade_partner_o_P;    // To communicate ...
  KK_inPort<int> *trade_partner_i_P;     // ... with trade partner
  Result make_Answer(TradeUnit*, Customer_id);
  void test_termination(TradeUnit*);
  void archive_result(Result*);
};

void BrokerAssistant :: start() {
  Agent::start();
  trade_P = new BB_Port<TradeUnit>(this, nV("trade"), INF);
  while (continue_p()) {
    // Get Trade Unit from the BB
    char* myKey = catStr(int2intstr(transaction_id),
                         int2intstr(customer_id));
    tu = trade_P->get(myKey);
    // Exchange information with the trade partner
    trade_partner_o_P =
     new KK_outPort<int>(this,nV(int2intstr(tu.trade_id)),1,INF);
    trade_partner_i_P =
     new KK_inPort<int>(this,nV(int2intstr(tu.trade_id)),1,INF);
    ...          // Negotiate
    // Put result on result port
    answ = make_Answer(&tu, partner_id);
    archive_result(&answ);
    // Terminate if everything sold
    test_termination(&tu);
  }
  stopMe();
}
```

Fig. 9.6. The BrokerAssistant class declaration and the implementation of its start method.

When a successful transaction (the result of the agreement of two Broker Assistants) is issued, both Broker Assistants inform their committed customers by transmitting appropriate information on their result_P port (of type KK_outPort), and then terminate. On customer side, results about processed queries can be collected either directly or through a *Secretary Agent*

(see figure 9.7). At regular intervals, not fulfilled queries are eliminated by the Trade Manager; all involved entities are kept posted.

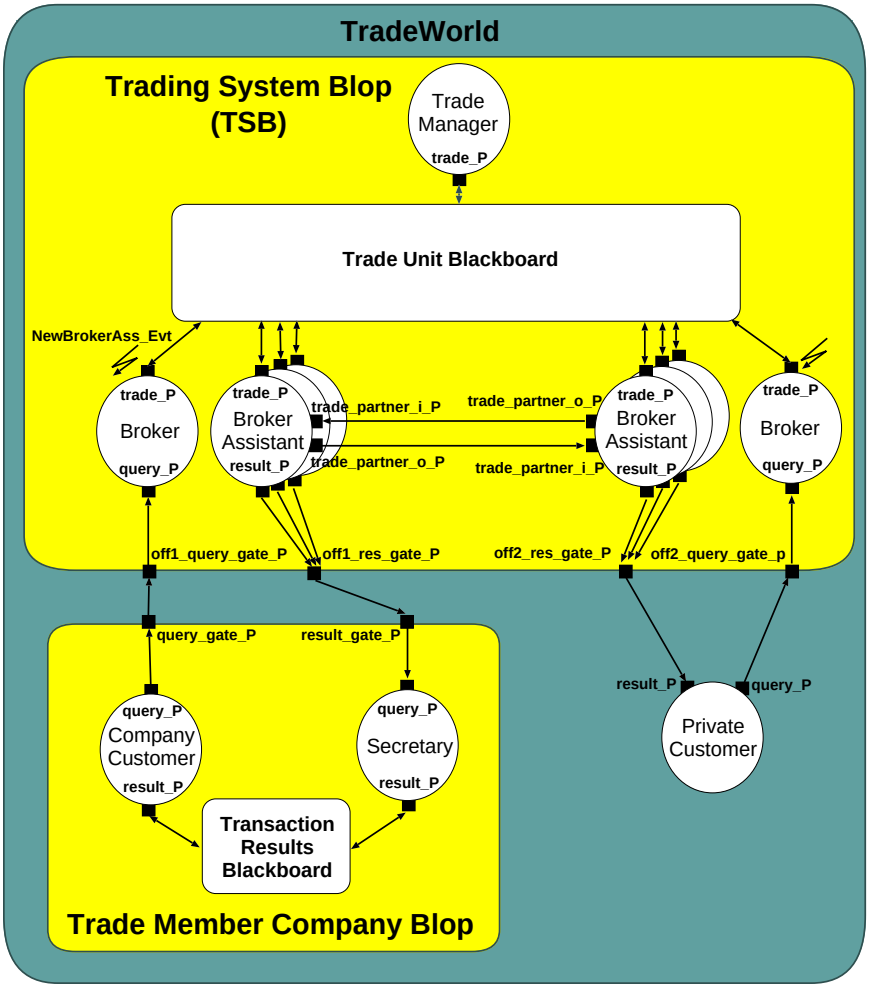

Fig. 9.7. Trading System with STL++.

10. Conclusion

The present work has focused on the use of coordination models and languages in the field of multi-agent systems. More precisely, it has thoroughly described a general coordination model named ECM and its instance STL++ in order to support the design and the implementation of simulated embodied agents' systems.

In chapter 2, we have first described what are autonomous agents and multi-agent systems, and we have maintained an explicit integration of the interaction setup in the modeling of a multi-agent system and have proposed to distinguish between objective and subjective coordination as the management of objective and subjective dependencies in a multi-agent system. Then, after having introduced embodied simulated autonomous agents systems, our target class of systems, we have proposed a specific approach to the implementation of multi-agent systems whose main benefit is an explicit integration of the objective coordination dimension in the design and implementation of a multi-agent system. To that aim, we have sustained the use of coordination models and languages (from the fields of concurrent and distributed computing, programming languages and software engineering) in order to support the design and the implementation of multi-agent systems. Hence, in chapter 3, we have proceeded with the review of existing coordination models and languages on the basis of an established taxonomy that classifies models as data-driven or process-oriented. We have, furthermore, presented several hybrid models, some of them being also used for multi-agent systems. Finally, we have analyzed the prerequisites which we have required for a coordination model and for a language to be used for our target class of multi-agent systems.

In chapter 4, we have presented the ECM model of coordination. Its main benefits, in regard to our requirements exposed in chapter 3, are the following:

- Thanks to blops, ECM allows a grouping mechanism that permits the encapsulation of a set of agents, local communication within the created space, and hierarchical structuring means for an environment.
- ECM processes support the realization of a basic autonomy, because they are conceived as blackboxes, which also permits a pro-active behavior. Process data is private and, therefore, not accessible to other entities.

M. Schumacher: Objective Coordination in MAS Engineering, LNAI 2039, pp. 121–124, 2001.

– Sensing and effecting on the environment are supported thanks to anonymous communication through ports.
– Basic communication paradigms are supported (stream, group and blackboard). Furthermore the setup of the communication means is ensured in a decentralized manner thanks to the implicit matching mechanism.

In the subsequent chapters we have presented the three existing ECM instances. In chapter 5, we have reviewed STL which is a separate language to be used in conjunction with a computation language realizing the coordinated processes. STL is intended at the implementation of multi-threaded applications in the context of network-based concurrent computing. STL showed important potentials for the implementation of multi-agent systems. This was the reason for the abstraction of STL's basic ideas towards a general coordination model (ECM) and the definition of a new coordination language, STL++. STL++ has the following benefits:

– It enhances dynamicity, enabling dynamic blop as well as agent creation and termination, and dynamic port creation and deletion, thus allowing reorganization of the connection patterns.
– It has a single language approach in order to facilitate dynamicity and to help uniformity in the programming.
– It is conceived in an object-oriented way, because every ECM abstraction has a corresponding class, and the operations on these abstractions are class methods.
– The semantics of each ECM construct is clearly specified, especially in what concerns port matching, deleted ports and consequent freed connections.

The use of STL++ has been illustrated by a classical tutorial example slightly modified. Furthermore, the core interfaces are explained in appendix A. Two main disadvantages of STL++ are its bad portability and its lack of agent migration mechanism between blops. This has led to the definition of a new prototype ECM instance named AGENT&CO presented in chapter 7. This coordination language has been realized as a JAVA mapping over an object request broker architecture.

Two case studies in STL++ have been explained in chapters 8 and 9. The former has shown the implementation of a generic model for simulated embodied agent systems. The latter has discussed another class of multi-agent systems by presenting the implementation of a trading system simulation.

The research presented in this book can be improved in several ways. We report a few of them:

Several enhancements of ECM and STL++ are possible such as:

– For the moment, data in blackboards is accessed using a string. A real associative access as in LINDA should be introduced.
– Programmable filters over the ports would enhance the implementation of sensors and effectors. Furthermore, an agent should be able to inquire the

state of a port in order to react to specific configurations, and possibly to change this state (for instance by modifying the port names).
- The semantics of the group communication should be generalized towards open groups.
- It would be worth to study the introduction of a broadcast mechanism. In any case, this broadcast should be contextualized in the own blop or in a fixed depth in the blop hierarchy.
- One common problem of process-oriented languages is the lack of aggregate operations on ports; this is also the case in ECM/STL++. In order to cope this weakness, the study of aggregate *read* and *get* should be undertaken.

The semantics of ECM and of STL++ have been presented in a semi-formal way, describing verbally each abstraction and the operations on it. It would be useful to define a theoretical framework in order to formalize ECM and its instances. This would facilitate new implementations and give means to compare the different ECM instances.

A graphical programming environment leading the development of an application could facilitate an implementation. Actually, as we have seen through the examples presented, several basic configurations can be depicted. Hence, a graphical environment could produce code skeletons that should be specialized by the programmer.

The object-oriented integration of ECM through STL++ could be further investigated. In particular, it would be useful to study to which extent the inheritance anomaly [MY93], [McH94] is present in STL++. Furthermore, in order to support general purpose implementations, one could define several organization patterns in form of classes that could be reused by specialization.

The work undertaken with AGENT&CO should be continued in order to study openness and mobility. Indeed, we consider that the mobility capacity of agents and the handling of resulting problems (such as security issues) are of the most promising research directions.

Software engineering of MASs is one of the greatest challenges in actual MAS research. The few existing methodology proposals are not complete and have to be enhanced on the basis of acquired experience. As proposed in this book, we consider the analysis of existing dependencies and the design and implementation of their management (the coordination) as key elements for the development of methodologies. Similar points of view are beginning to be shared by few recent works such as [MJL00], [Omi00], [ZJOW01]. We thus propose to complete our basic methodology in order to cope with a larger set of applications.

One possible emphasis in the engineering of MASs is the identification of *laws* ruling a MAS. We recognize at least two families of laws: i) *communication laws*, that regulate every communication event (e.g. filters, enhancement of exchanged messages, observation); ii) *environment laws*, that cope with the organization of the environment (e.g. data organization, consequences of

data density, physical constraints such as bandwidth or memory, load balancing, time dependencies). The liability of theses laws may be strong or weak, coupling them with penalties that limit access rights, increase control or constraint lifetimes. A coordination model should then be designed in such a way that it can express such laws. This may be done similarly to reactive blackboards using broadcasted events that indicate state changes, and defining laws that simply capture these events and react appropriately.

A. Core STL++ Interfaces

The present appendix is devoted to the core interface of STL++. The class hierarchy can be found in figure 6.2. World, blop and agent classes are first presented. Then come the port template classes, and the event and condition classes. Finally, macros and miscellaneous functions of STL++ are exposed.

A.1 World Class

`class WorldBlop`

Implements the world blop encompassing every other entity in an STL++ application.

`WorldBlop()`

Creates a world blop. Note that no creation macro is necessary for the world blop creation.

`void stopMe()`

Waits for every other activity in the world blop to stop, and terminates.

A.2 Blop Class

`class Blop`

`Blop()`

Creates a blop. In order to create an instance of a class inheriting from `Blop`, the macro `createBlop` must be called (see section A.7). If the creator of a blop b is another blop f, then b is created inside f; if the creator of b is an agent or an event, b is created inside their parent blop.

`virtual void start() = 0`

Pure virtual method that must be re-implemented. Defines the acting of the blop, which can create blops, ports, events and agents. This method is automatically called by the blop creation macro `createBlop`.

`void stopMe()`

Waits for every other activity in the blop to stop, and terminates.

M. Schumacher: Objective Coordination in MAS Engineering, LNAI 2039, pp. 125–131, 2001.

A.3 Agent Class

`class Agent`

`Agent()`
Creates an agent. In order to create an instance of a class inheriting from `Agent`, the macro `createAgent` must be called (see A.7). If the creator of an agent a is a blop f, then a is created inside f; if the creator of a is another agent or an event, a is created inside their parent blop.

`virtual void start() = 0`
Pure virtual method that must be re-implemented. Defines the acting of the blop, which can create blops, ports, events and agents. This method is automatically called by the agent creation macro `createAgent`.

`void stopMe()`
Stops the thread of execution and terminates.

A.4 Port Template Classes

All port classes are template classes with one template argument `DataT` indicating the type of the data that can flow through a port. For each basic type of ports, two classes have been defined, one for agents and the other one for blops; however one exception exist, namely there are no blop blackboard ports. Blop and agent ports differentiate themselves in their first actual parameter of their constructor; it indicates the creator of the port by a pointer on a `Blop` or an `Agent` object. In practice, the pointer `this` is passed as argument by the creator.

For all port classes, if a data item is posted on a port that is not connected, it is bufferized until a connection is made available; when the port is connected, all bufferized data items are automatically forwarded to the connection.

All ports are descendants of the `Port` class. This class has basic member functions for testing the state of the port and attaching an event to it:

`template<class DataT> class Port`

`int getConnNbr()`
Gets the number of connections a port is bound to.

`bool live_p()`
Returns true if the lifetime of the port has passed, else returns false.

`void attachEvent(Event *e)`
Attach the event `e` to the port.

A.4.1 KK-Stream Ports

Apart from the first argument indicating the creator (blop or agent), the constructors of all KK-Stream ports have identical formal parameters. `theNames` is a vector of `char*` indicating the names of the port; `satF` defines the saturation; and `lifetF` indicates the lifetime of the port.

```
template<class DataT> class KK_inPort
    KK_inPort(Agent *a, VectorOfNames *theNames, int satF,
              int lifetF)
```
Creates an in-going KK-Stream port belonging to an agent `a`.
```
    DataT get()
```
Returns a data item of type `DataT` if it is available in the port. If no data item is present or if the port is not connected, the method blocks until one data item is available; it then returns the data item.
```
template<class DataT> class KK_outPort
    KK_outPort(Agent *a, VectorOfNames *theNames, int satF,
               int lifetF)
```
Creates an outgoing KK-Stream port belonging to an agent `a`.
```
    void put(DataT putMessage)
```
Posts a data item of type `DataT` on the port. The method is non-blocking.
```
template<class DataT> class KK_Blop_inPort
    KK_Blop_inPort(Blop *b, VectorOfNames *theNames, int satF,
                   int lifetF)
```
Creates an in-going KK-Stream port belonging to a blop `b`. No `get` method is available, because blop ports only serve as gateways between the inside and the outside of a blop.
```
template<class DataT> class KK_Blop_outPort
    KK_Blop_outPort(Blop *b, VectorOfNames *theNames, int satF,
                    int lifetF)
```
Creates an outgoing KK-Stream port belonging to a blop `b`. No `put` method is available, because blop ports only serve as gateways between the inside and the outside of a blop.

A.4.2 S-Stream Ports

Apart from the first argument indicating the creator (blop or agent), the constructors of all S-Stream ports have identical formal parameters. `theNames` is a vector of `char*` indicating the names of the port; `satF` defines the saturation; and `lifetF` indicates the lifetime of the port.

```
template<class DataT> class S_inPort
    S_inPort(Agent *a, VectorOfNames *theNames, int satF,
             int lifetF)
```
Creates an in-going S-Stream port belonging to an agent `a`.

`DataT get()`
Returns a data item of type `DataT` if it is available in the port. If no data item is present or if the port is not connected, the method blocks until one data item is available; it then returns the data item.

```
template<class DataT> class S_outPort
```

```
S_outPort(Agent *a, VectorOfNames *theNames, int satF,
          int lifetF)
```
Creates an outgoing S-Stream port belonging to an agent `a`.

`void put(DataT putMessage)`
Posts a data item of type `DataT` on the port. The method blocks until the data item has been read by another agent.

```
template<class DataT> class S_Blop_inPort
```

```
S_Blop_inPort(Blop *b, VectorOfNames *theNames, int satF,
              int lifetF)
```
Creates an in-going S-Stream port belonging to a blop `b`. No `get` method is available, because blop ports only serve as gateways between the inside and the outside of a blop.

```
template<class DataT> class S_Blop_outPort
```

```
S_Blop_outPort(Blop *b, VectorOfNames *theNames, int satF,
               int lifetF)
```
Creates an outgoing S-Stream port belonging to a blop `b`. No `put` method is available, because blop ports only serve as gateways between the inside and the outside of a blop.

A.4.3 Blackboard Ports

Blackboard ports, which can only be created by agents, have a fixed saturation of one; this means that a port can be connected only to one blackboard connection.

```
template<class DataT> class BB_Port
```

```
BB_Port(Agent *a, VectorOfNames *theNames, int lifetF)
```
Creates an blackboard port belonging to an agent `a`, with the names of the vector `theNames` and a lifetime set to `lifetF`.

`DataT get(char* key)`
Returns a data item of type `DataT` that matches the key `key`. If several matching are possible, one data item is chosen nondeterministically. If no matching is possible or if the port is not connected, the method blocks until one matching data item is found; it then returns the data item.

`void put(char* key, DataT putMessage)`
Posts a data item of type `DataT` with the key `key`. The method is non-blocking.

A.4.4 Group Ports

Apart from the first argument indicating the creator (blop or agent), the constructors of all group ports have identical formal parameters. `theNames` is a vector of `char*` indicating the names of the port; `satF` is the saturation; and `lifetF` indicates the lifetime of the port.

```
template<class DataT> class Group_Port
```

`Group_Port(Agent *a, VectorOfNames *theNames, int satF, int lifetF)`
: Creates a group port belonging to an agent `a`.

`DataT get()`
: Returns a data item of type `DataT` if it is available in the port. If no data item is present or if the port is not connected, the method blocks until one data item is available; it then returns the data item.

`void put(DataT i)`
: Posts a data item `i` of type `DataT` on the port. The data item is forwarded to every member of the group. The method is non-blocking.

```
template<class DataT> class Group_Blop_Port
```

`Group_Blop_Port(Blop *b, VectorOfNames *theNames, int satF, int lifetF)`
: Creates a group port belonging to a blop `b`. Neither `get` nor `put` methods are available, because blop ports only serve as gateways between the inside and the outside of a blop.

A.5 Event Class

```
class Event
```

`Event(Condition *c, int lifetime)`
: Creates an event object with lifetime `lifetime`. The lifetime is decremented each time the event is launched. A pointer `c` to a condition object indicates under which condition (see section A.6) the event is triggered.

`virtual void launch() = 0`
: Virtual method that must be re-implemented. Defines the acting of the event.

`int live_p()`
: Returns true if the lifetime has passed; otherwise false.

A.6 Condition Classes

Condition objects are used in event constructors (see section A.5). All condition constructors have no arguments, with the exception of the `MsgHandledCond` class.

`class UnboundCond`
`UnboundCond()`
Creates a condition object that is fulfilled if the port is not connected at all.

`class SaturatedCond`
`SaturatedCond()`
Creates a condition object that is fulfilled if the port is bound to all its potential connections.

`class MsgHandledCond`
`MsgHandledCond(int n)`
Creates a condition object that is fulfilled if `n` messages have passed through the port.

`class AccessedCond`
`AccessedCond()`
Creates a condition object that is fulfilled whenever the port has been accessed by a primitive.

`class PutAccessedCond`
`PutAccessedCond()`
Creates a condition object that is fulfilled whenever the port has been accessed by the `put` primitive.

`class GetAccessedCond`
`GetAccessedCond()`
Creates a condition object that is fulfilled whenever the port has been accessed by the `get` primitive.

`class TerminatedCond`
`TerminatedCond()`
Creates a condition object that is fulfilled whenever the port lifetime is over.

A.7 Macros and Miscellaneous Functions

`declareRunnable(class_name)`
Macro that declares a blop or an agent of class `class_name` as runnable.

`defineBlop(class_name, par_types ...)`
Macro that declares class `class_name` with parameters of types `par_types` as a blop class.

`defineAgent(class_name, par_types ...)`
Macro that declares the class `class_name` with parameters of types `par_types` as an agent class.

`createBlop(class_name, par_list ...)`
Macro that creates a blop of class `class_name` with parameters `par_list`. If the creator is a blop f, the new blop is created inside f; if the creator is an agent or an event, the new blop is created in the blop of the creator. The macro creates an instance of the class and launches its `start` method.

`createAgent(class_name, par_list ...)`
: Macro that creates an agent of class `class_name` with parameters `par_list`. If the creator is a blop f, the new agent is created inside f; if the creator is an agent or an event, the new agent is created in the blop of the creator. The macro creates an instance of the class and launches the `start` method.

`void STLppInit()`
: Initializes the STL++ system.

`void STLppTerminate()`
: Waits for the world blop to terminate and shuts down the STL++ system.

`VectorOfNames* nV(char *n, ...)`
: Creates a vector of names of type `VectorOfNames` from the list of names given in the parameter list of type `char*`. `VectorOfNames` is identical to `vector<char*>`, `vector` being defined in the C++ Standard Template Library [Str97].

B. STL Code Example

In order to allow a comparison with STL++, this appendix exposes the implementation in STL of the sieve of Eratosthenes, as proposed in [Kro97]. The realization, which follows an analogous organization as figure 6.1, has two parts: a computation part (figure B.1), and an implementation part (figure B.2). Further details can be found in the cited reference.

```
1  void sieve(THREAD_ENV *p, P2P_in in, P2P_out out, prim) {
2    const int tag = 2;
3    IntTempl t, n;
4    Msg Prime(p), Number(n);     /* The messages to receive */
5
6    in.get(tag, Prime);          /* Get our prime */
7    prim.put(tag, Prime);        /* Tell source process */
8    in.get(tag, Number);         /* Get a number from pipeline */
9
10   while(n % p == 0) {          /* Sieve until a new candidate */
11     in.get(tag, Number);       /*                      is found */
12   }
13   Msg Div(n);                  /* A new candidate */
14   out.put(tag, Div);           /* forward it in pipeline */
15
16   while(1) {                   /* Do your usual work, sieve */
17     in.get(tag, Number);       /* Get candidate */
18     if(n % p != 0) {
19       out.put(tag, Number);    /* Forward it */
20     }
21   }
22 }
```

Fig. B.1. Sieve of Eratosthenes in STL: computation part.

M. Schumacher: Objective Coordination in MAS Engineering, LNAI 2039, pp. 133–134, 2001.

```
1  blop world {
2    process source(out, prim) {              // Source process
3      P2P->out<"SOURCE">;                    // Output port
4      P2P<-out<"PRIM">;                      // Input port
5      func source;                           // Thread entry point
6    }
7    process sieve_p(in, out, prim) {
8      P2P<-in<"SOURCE">;                     // The definition
9      P2P->prim<"PRIM">;                     //    of the filter
10     P2P->out<"SOURCE">;                    //  process
11     func sieve;
12   }
13   create process sieve_p s;                // Initial filter process
14   event new_sieve() {
15     create process sieve_p n;              // New filter process
16     when unbound(n.out)                    // Re-install event
17       then new_sieve();
18   }
19   when unboud(s.out) then new_sieve();     // Attach event to
20   create process source q;                 // out port of s
21 }
```

Fig. B.2. Sieve of Eratosthenes in STL: coordination part.

C. Linda, Gamma and Manifold Code Examples

In order to allow a comparison with STL++, this appendix presents implementations in Linda (figure C.1), Gamma (figure C.2) and Manifold (figure C.3) of the Restaurant of Dining Philosophers (see section 6.7) as presented in [ACH98]. Further details can be found in the cited reference.

```
#define TRUE 1

philosopher(int i) {
  while(TRUE) {
    think();
    in("meal_ticket");
    in("fork", i);
    in("fork", (i+1)%5);
    eat();
    out("fork", i);
    out("fork", (i+1)%5);
    out("meal_ticket");
  }
}

real_main() {
  int i;
  for(i=0; i<5; i++) {
    out("fork", i);
    eval(philosopher(i));
    if(i<4)
      out("meal_ticket");
  }
}
```

Fig. C.1. The Restaurant of Dining Philosophers in Linda.

```
("fork", i), ("fork", j) -> ("eat", i) <= j=(i+1)%5
("eat", i) -> ("fork", i), ("fork", (i+1)%5) <= true
```

Fig. C.2. The Restaurant of Dining Philosophers in Gamma.

M. Schumacher: Objective Coordination in MAS Engineering, LNAI 2039, pp. 135–136, 2001.

```
#define WAIT (preemptall, terminated(self))

event request, done.
manner Eat(process, process, process) import.
manner Think(process) import.
manner GetTicket() import.
manner ReturnTicket() import.

export Fork() {
  begin: while true do {
           begin: WAIT.

           request.*phil & *ready.*phil: {
             save *.
             begin: (raise(ready), WAIT).
             done.phil:.
             }.
         }.
}

export Philosopher() {
  event ready.

  begin: while true do {
           begin: Think(self);
                  GetTicket();
                  (raise(request, ready), WAIT).

           ready.*lfork & ready.*rfork: Eat(self, lfork, rfork).

           end: raise(done);
                ReturnTicket().
         }.
}
```

Fig. C.3. The Restaurant of Dining Philosophers in Manifold.