

Contents

1	Autonomous Agents	3
1.1	A brief overview of Artificial Intelligence's state of the art	3
1.1.1	Classical Artificial Intelligence	3
1.1.2	Behavior-based Artificial Intelligence	3
1.1.3	Conclusion	3
1.2	Behavior-based Artificial Intelligence and Autonomous Agents	3
1.2.1	Interactions and Emergence	4
1.2.2	Autonomy	4
1.2.3	Embodiment	4
1.3	Our Autonomy-based Multi-Agent System	5
1.3.1	A generic model	5
1.3.2	Typical Application	5
1.3.3	Major problem inherent to a serial implementation	6
1.3.4	Major constraints for a distributed implementation	7
2	Coordination Theory	7
3	Coordination Models	9
4	The Coordination Model of STL	11
4.1	Blops	13
4.2	Processes	13
4.3	Ports	14
4.3.1	Port Attributes	14
4.3.2	Basic Port Types	15
4.3.3	Variations of the Basic Port Types	16
4.3.4	Port Matching	16
4.3.5	Static Ports	17
4.3.6	Dynamic Ports	18
4.4	Connections	18
4.4.1	Connections as Streams	18
4.4.2	Connections as Blackboards	18
4.5	Events	18
4.6	Primitives	19
4.6.1	Operations on dynamic Ports	19
4.6.2	Operations to transfer Data	20
4.6.3	Operations on Events	20
5	Our Autonomy-based Multi-Agent System in STL	21
5.1	Introduction	21
5.2	Global Structure	21
5.2.1	Blop <i>se</i>	22
5.2.2	Ports of a Blop	22
5.2.3	<i>initAgent</i> Process, <i>newAgentEvt</i> Event	22
5.2.4	<i>agent</i> Process	23
5.2.5	<i>subEnv</i> Process	23
5.2.6	The <i>taxi</i> Process	23
6	Conclusion	24

Coordinating Autonomous Entities*

O. Krone, F. Chantemargue, T. Dagaëff, M. Schumacher, B. Hirsbrunner

University of Fribourg, Computer Science Department, PAI group

Chemin du musée 3, Pérolles, CH-1700 Fribourg, Switzerland

E-mail: {Firstname.Lastname}@unifr.ch

URL: <http://www-iiuf.unifr.ch/pai>

Abstract

This paper describes our work, which aims at providing a generic coordination framework for implementing distributed multi-agent systems made up of autonomous agents. Three major parts can be identified: the first one is devoted to the presentation of our generic autonomy-based multi-agent system and points out its most important concepts and constraints in order to shape the coordination model. The second part describes in detail STL, our coordination model, and related tools. The last part shows a preliminary distributed implementation of our autonomy-based multi-agent system using STL's primitives, in the case of a commonly addressed application, viz. a collective robotics simulation.

Keywords: Multi-Agent System, Autonomy, Collective Robotics, Coordination, Distributed Systems.

Introduction

Coordination and *Autonomous Agents* constitute two major domains of *Computer Science*. Works coming within *Coordination* encompass conceptual and methodological issues as well as implementations in order to efficiently help expressing and implementing distributed applications. Works coming within *Autonomous Agents* draw inspiration from natural systems and are aimed at designing entities exhibiting human or animal-like behavior at an individual and/or a collective level. Intended to capitalize on the co-existence of distributed entities, models such as *Multi-Agent Systems* are oriented towards interactions, collaborative phenomena and autonomy.

Although a conceptual parallelism of the agents is inherent to multi-agent models, most of their implementations are serial. Firstly, as the parallelism of the agents is underlying in the model, it is usually believed to be implicitly taken into account whatever the implementation (serial or distributed). Secondly, existing coordination languages, models and programming environments suffer from limitations concerning mechanisms to coordinate autonomous agents, so that a distributed multi-agent implementation is complex and burdensome. The present work is aimed at providing a coordination framework for distributed multi-agent systems made up of autonomous agents.

This paper is organized as follows. Section one gives at first a review of *Autonomous Agents* in the context of *Artificial Intelligence*, before introducing the most relevant concepts of *Autonomous Agents*. Then our generic autonomy-based multi-agent system will be presented and the major constraints to be taken into account for a distributed implementation will be pointed out. Section two presents a brief review on *Coordination* theory. Section three gives an overview of most common existing coordination models. Section four describes in detail our coordination model (STL) and appropriate coordination

*Part of this work is financially supported by the Swiss National Funds for Scientific Research, grants 21-43558.95 and 21-47262.96

language. Section five presents a preliminary distributed implementation of our autonomy-based multi-agent system illustrating a collective robotics simulation, in order to show the appropriateness of STL. In the last section, we draw some conclusions from this work and outline future works.

1 Autonomous Agents

1.1 A brief overview of Artificial Intelligence's state of the art

Artificial Intelligence (AI) aims at synthesizing intelligence in artifacts. Two families of approaches exist, disagreeing in their notion of what intelligence actually means.

1.1.1 Classical Artificial Intelligence

On the one hand, *Classical AI* (also referred to as *Knowledge-based AI* or *Top-Down AI* in the literature) defines intelligence as the capacity to form and manipulate internal representational models (knowledge) of the world. Therefore, *Classical AI* emphasizes the realization of systems that possess a certain amount of knowledge about some problem domain (so-called expert systems) that allows them to reason within this domain. Hence, the central area of research within this field is the problem of knowledge representation and knowledge engineering: how to extract a human-knowledge and represent it in an appropriate form for a machine to reason with? *Classical AI* succeeds in realizing systems with in many cases an impressive performance/expertise in a narrow domain, such as medical diagnosis, theorem proving, chess playing, etc. However, these systems suffer from the problem of lacking ability to relate internal representations/knowledge to the external world [12] and they tend to utterly fail when facing problems even slightly outside their domain of expertise [29], [59]. Most common drawbacks inherent to *Classical AI* are known as the *Frame Problem* (problem of maintaining a model of the real world [50]) and the *Grounding Problem* (problem of relating the elements of the representation to the sensory information [27]).

1.1.2 Behavior-based Artificial Intelligence

Behavior-based AI [38] (also referred to as *Bottom-Up AI* or *New AI* in the literature) on the other hand considers intelligence as a biological feature [41] (this notion is often referred to as *Enactivism* or *Enaction* [58]): intelligence should be considered an agent's capacity to interact with its environment, rather than represent/model it internally. Therefore, the emphasis is led on adaptivity. *Behavior-based AI* is concerned with systems exhibiting life-like features, in particular autonomous agents that behave within some environment. Moreover, in analogy to living systems, *Behavior-based AI* also studies intelligence as result of adaptation at the group level as well as species level (evolutionary adaptation/learning).

1.1.3 Conclusion

In conclusion, *Bottom-Up AI* consists in starting off with agents, whose competences are very basic, e.g. of sensory-motor coordination type, and in achieving behavior complexity through the interactions of the agents in the environment. Therefore, the emphasis is led on incremental learning/adaptation and development of the system over time. Conversely, *Top-Down AI* consists in modeling high level cognitive capacities such as planning. Thus, *Top-Down* systems are rather static and often unable to improve/adapt themselves.

1.2 Behavior-based Artificial Intelligence and Autonomous Agents

Autonomous agents are considered to be embodied systems, designed to fulfill internal or external goals by their own actions in continuous long-term interaction with the environment (possibly unpredictable and dynamical) in which they are situated [8]. This position is opposed to the one of *Classical AI*, where

expert systems are disembodied programs, whose only connection to their environment is their user. Thus, in *Autonomous Agents*, the emphasis is focused on the interactions between the system and the environment.

1.2.1 Interactions and Emergence

Dealing with interactions leads naturally to the very concept of emergence of behavior and/or functionality. Emergence offers indeed a bridge between the necessity of complex and adaptive behavior at a macro level (the one of an observer) and the mechanisms of multiple competences and situation-based learning at a micro level (the one of the agents). A system's behavior can be considered emergent if it can only be specified using descriptive categories which are not necessary to describe the behavior of the constituent components. Steels in [54] adds that an emergent behavior leads to emergent functionality if the behavior contributes to the system's self-preservation and if the system can build further upon it. Numerous works deal with emergence: some are concerned with its formalization [45], some others are concerned with bringing to the fore some experimental issues [20]. For instance, in [47], a robot follows a corridor (emergent behavior) by turning right when the wall is on the left and by turning left when the wall is on the right and by moving straightforward otherwise (three particular behaviors). Another example [23] is given by our system, which will be described in the next section: it illustrates the emergence of cooperation in the framework of mobile robotics. A collection of agents stack up, at a location which is the result of the agents' interactions, some objects that are distributed in the environment (emerging behavior). Every agent's behavior consists in taking and dropping an object according to a probability law.

1.2.2 Autonomy

In *Autonomous Agents*, two types of autonomy are commonly pointed out [60]: operational autonomy and behavioral autonomy. Both types agree in the idea that automaticity is necessary to autonomy. According, to Steels in [55], an agent, to be autonomous, must first be automatic: it must be able to operate in an environment, to sense this environment and to impact in ways that are beneficial to the agent and to the tasks that are crucial to its further existence. In [49], operational autonomy is defined as the capacity to operate without human intervention, without being remotely controlled. In [55], behavioral autonomy supposes that the basis of self-steering originates in the agent's own capacity to form and adapt its principles of behavior: an agent, to be behaviorally autonomous, does not only need the freedom to behave/operate without human intervention (operational autonomy), but further the freedom to have formed (learned or decided) its principles of behavior on its own (from its experience), at least partly. There are two broad classes for designing autonomy-based multi-agent systems: the engineering approach for operational autonomy, and learning approaches for behavioral autonomy. Engineering approaches are based on a functional decomposition of the control task, typically in some form of finite state automaton. The most commonly known approach of this type is the subsumption architecture [12]. Learning approaches consist in endowing each agent with some learning capabilities. Thus, the agent can adapt its behavior (self-learning through artificial neural networks, [47]) or inherit it from earlier generations of agents (evolutionary learning).

1.2.3 Embodiment

Autonomous agents are embodied systems by definition, however there are different forms of embodiment. Some works, for instance [12], [13], consider *physically embodied agents* interacting with their environment by means of sensory perception and motor control (case of robotic forms of intelligence). Some other works consider *simulated embodied agents* interacting with realistic environments [9], [23], [37], [42] (case of simulated robotic form of intelligence). A simulation is indeed generally required prior to some robotic application. It permits certain properties to be grasped, feasibility or relevance of some approaches to be checked through an intensive number of experiences which would be prohibitive (both

in time and money) to realize in a real world. In some other works, embodiment is interpreted rather as *situatedness* or *embeddedness*, that is, the property of existing within and interacting with an environment [39]. Agents of this type, referred to as software agents (see [48] for a review), lack body, sensors, etc, but unlike *Classical AI* programs directly interact with complex software environments.

1.3 Our Autonomy-based Multi-Agent System

1.3.1 A generic model

We propose a model for autonomy-based multi-agent systems which is general enough to be applied to numerous domains: it is composed of an *Environment* and a list of *Agents*. The *Environment* encompasses a list of *Cells* and a set of *Objects* which will be manipulated by the agents. Every *Cell* has a list of *Neighbour Cells*, which implicitly sets the discrete topology and contains *Objects*. This way of encoding the environment allows the user to cope with any type of topology, be it regular or not, since for every cell the number of neighbours can be specified. Note that a cell can contain a region made up of a set of continuous points, e.g. for simulating an area with real coordinates rather than discrete.

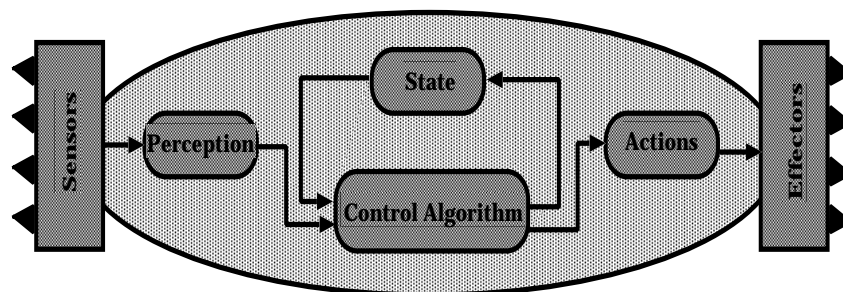


Figure 1: Architecture of an agent

The architecture of an agent is displayed on figure 1. An agent possesses some sensors to perceive the world within which it moves, and some effectors to act in this world, making it falling into the class of *simulated embodied agents*. Note that this architecture complies as well with the prescriptions of the class of *physically embodied agents*. The implementation of the different modules presented on figure 1, namely *Perception*, *State*, *Actions* and *Control Algorithm* depends on the application and is the user's responsibility. In the *Perception* module, the designer specifies the type of perception of the agent, e.g. if the agent perceives only the number of objects on the cell on which it stands. The *State* module encompasses the private stuff of the agent, e.g. whether it carries or not an object, its seed value for its random generator, its strain or whatever. The *Actions* module typically consists of the basic actions the agent can take, e.g. move to next cell, take an object, drop an object, etc... The *Control Algorithm* module is particularly important because it defines the type of autonomy of the agent: it is precisely inside this module that the designer decides whether to implement an operational autonomy or a behavioral autonomy. For instance, a very basic operational autonomy consists of randomly choosing the type of action to take. A behavioral autonomy would consist of implementing some learning capabilities, e.g. by using an adaptive neural network.

1.3.2 Typical Application

We illustrate with an application in the framework of mobile robotics and more specifically collective robotics. Many robotics applications rely indeed on the collaboration within a group of robots, where the functioning of the whole group depends on the competences of each individual, as well as on the interactions between individuals. Our application tackles a quite common problem in collective robotics which is still given a lot of consideration: agents (an agent simulates the behavior of a real robot) seek

for objects distributed in their environment, and we would like them to stack all objects, like displayed in figure 2. However, the way we solve this problem is quite uncommon: the innovative aspect of our approach rests indeed on a system integrating operationally autonomous agents, that is, every agent in the system decides by itself which action to take.

Several serial versions of this simulation have been already successfully implemented on a sun workstation (under Solaris, in ObjectiveC language using SWARM [36]), depending on the type of module implementations. This simulation exhibited the emergence of properties in the system, such as cooperation yielded by the recurrent interactions of the agents; agents cooperate to achieve a task without being aware of it, and without requiring any type of cooperation protocol. Further details about this simulation and outcomes can be found in [20], [18], [19], [23]. An implementation in a real world using real mobile robots (khepera robots [44]) is currently being developed.

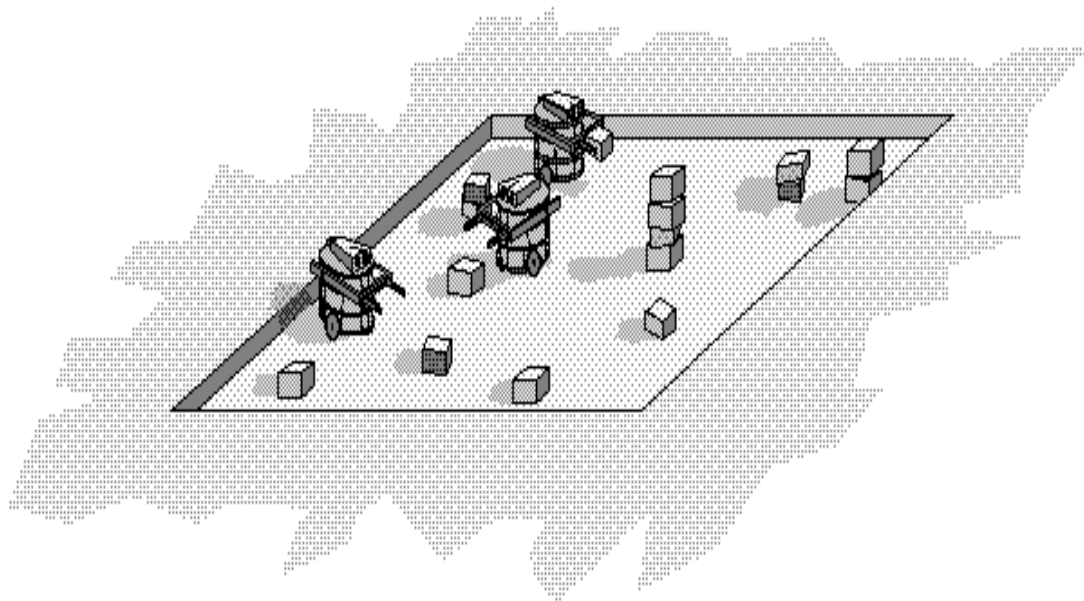


Figure 2: A collective robotics application: stacking objects

Our generic model may as well be applied to the framework of parallelism in computer science to address the problem of collecting results yielded by numerous nodes in a distributed architecture or to address load balancing problems, each cell being a node and objects being results or load for instance.

1.3.3 Major problem inherent to a serial implementation

Let us now point out the following remarks about our autonomy-based multi-agent model from a conceptual point of view. First of all, agents co-exist (in space and time) and share a common environment. Secondly, agents can move, act in parallel in the environment, each one having its own freedom for deciding: for instance it is clear that an agent can move while carrying an object in the north area of the environment, whereas in the meantime another agent will drop an object in the south area. This implies a certain independence between agents in the system and between areas of the environment. Every agent has its own time or temporality, which is peculiar to its identity: the motion of the agent roaming in the north area is indeed most unlikely to be identical to the one of the agent roaming in the south area. At least, this is what we have in mind when developing the model and this is how we would like the implementation of the model to behave.

However, in the serial implementation it does not work exactly that way. Every agent in the list is given the processor in turn: the scheduling is performed in a round robin manner. If the list conveys

quite well the notion of co-existence in space, it however does not fully render the notion of co-existence in time. The whole system possesses indeed a single time scale, whereof every agent is given a slice in turn. Thus, the functioning of the global system is not exactly what could be conceptually expected. Eventhough the system showed that it performed well and exhibited some interesting properties, the implementation did not really match with the model.

As our multi-agent system is aimed at addressing problems which are naturally distributed (both in space and time), a distributed implementation appears to be better suited than a serial implementation. Simulations have indeed to be as closest as possible to the reality they simulate, in order to avoid drawing misleading conclusions. Note that an attempt to minimize the problem of time for the agents in the serial implementation would consist of endowing each agent with its own time slice (which may possibly change over time, which may be set randomly by the agent, etc...). It however will not fix the problem, it will just allow the user, who has not a distributed implementation, to simulate different time scales and to have an estimate of the impact.

1.3.4 Major constraints for a distributed implementation

Our very aim is to be able to express our autonomy-based multi-agent system on a distributed architecture in the most natural way. For that, we need an appropriate software platform in which the implementation would be the image of the model (the concepts) with the highest fidelity. We need some tools allowing us to coordinate our autonomous agents (since they interact in their environment) without modifying their structure and their behavior, like for instance reducing their degree of autonomy.

If the problem of spatially co-existing agents is straightforward to be implemented on a distributed system (computation units are already spatially distributed), the problem of co-existing in time (temporality for each agent in the system) should be given a great consideration. In distributed systems, time is usually subordinated to space. Generally, distributed applications capitalize on the property of spatial distribution of services, e.g. two independent processes that can be executed in parallel on two nodes in order to improve the performance. Distributed applications are most of the time made up of processes which can run simultaneously on different nodes, and which have to be synchronized by message exchanges in order to collect some (intermediate) results. Such applications do not fully convey the notion of proper time for each process, and moreover they do not aim at: thus, processes are embedded in a static coordination structure, where the slower process imposes its time (temporality) to the system (faster processes have indeed to wait for the slower process when synchronizing each others). The problematics in our application is rather different. As previously stated, temporality features identity and autonomy: an entity is not anymore autonomous when its temporality is subsumed to an external time scale.

We have therefore to identify some flexible coordination patterns that will not alter every agent's temporality and behavior, and we have to develop corresponding tools and primitives. Our platform must be able to render to some extent the independence in space and time, without being interfered by coordination mechanisms exclusively embedded for a purpose of implementation. It seems clear that the environment has to be distributed. Two types of mechanisms will have to be developed: some in order to cope with agents crossing borders between sub-environments (of course this should be achieved transparently to the user, it should be part of the software platform), and others in order to cope with data consistency (e.g., updating the number of objects on a cell).

2 Coordination Theory

The term coordination theory comprises a new discipline of science. Depending on the type of a parallel algorithm, state of the art programming models for parallel computing, such as (distributed) shared memory models, data and task parallelism, and parallel object oriented models (for an overview see [53]), make it possible to express parallel programs efficiently. However, all models suffer from some limitations concerning a clean separation of the computational part of a parallel application and the

Coordination mechanism	Dependency	Coordination process
Temporal location	Simultaneity constraints	Scheduling
Organizational roles	Task/Subtask	Adaptive behavior according to designated role
Reducing uncertainty in dynamic environment	Influence from environment	Use of laws and roles

Table 1: Basic Coordination Mechanisms.

“glue” that coordinates the overall distributed program. Therefore we need additional methods that describe the interdependencies of parallel activities in a clean way. This “glue” should be part of the application.

To give a first idea of the characteristics of this “glue”, one can look at the example of bees working in a bee hive, an inherent parallel “application”. Three important aspects can be seen: Firstly, the single individual, the bee, is of little significance, only the whole mass of bees produces the final result. Secondly, all bees are working towards a common goal, even though the single bee is not aware of this fact. Thirdly, bees have to coordinate their work somehow, they do this by exchanging signals. The following sections will show that these characteristics of a coordinated distributed application are valid for parallel programs too.

To define such a “glue”, Malone [40] introduced a new theory called *coordination theory*. Principles developed in this theory draw their inspiration not only from computer science, but from other disciplines, such as organization theory, operations research, economics, linguistics, ethology (see the example above), and psychology as well. One of the main objectives of this theory is to transfer knowledge about coordination from one discipline to another. So to speak, it would be desirable to apply coordination mechanisms known from for example organization theory in computer science.

In order to take advantage of such a transfer, one has to develop abstract coordination mechanisms which are valid in different disciplines. The design of such a coordination mechanism reveals an important fact in coordination theory. Surprisingly coordination is noticed if it is not there, that is, it is the *lack* of coordination that is most likely to be recognized. Unbalanced workload on a cluster of workstations, redundant activities, and not obeyed scheduling constraints are all examples of such a lack of *concurrent* coordination. Uncoordinated programs originate in fundamental problems of distributed computing. Redundant activities arise if agents have an inconsistent view of the whole application, unbalanced workstations are the result of an incomplete exchange of load information between the workstations, and finally not obeyed scheduling constraints can have their origin in a dynamic change of the environment or in prerequisites under which they were made.

In despite of these difficulties, Decker [24] identified three basic coordination mechanisms which can be observed in several disciplines. They are summarized in Table 1.

- *Temporal location*: is a mechanism in which schedules, plans, appointments are used to reduce uncertainty due to an incomplete view;
- *Organizational roles*: are applied to coordinate transactions according to predefined structures; agents behave according to their assigned role. For example, in a typical master/slave application, the master agent behaves differently from the slave agent;
- *Reducing uncertainty in the dynamics of the environment*: is mostly done by the introduction of laws. For example traffic lights (stop on red), wehre one does not have to negotiate at each crossing with every driver about who has the right of the way.

Coordination theory is a new discipline and there are still various aspects which merit a deeper investigation [40], such as

- How general are coordination processes? Can we identify coordination patterns generic enough

to be used in very different situations? What relation exists between general problem solving heuristics and general heuristics for coordination?

- Can we develop coordination metaphors which are totally independent from particular tasks and situations, or is every coordination process a special case of a particular coordinated situation?
- Is it possible to define a degree of coordination? Can we measure how coordinated an application is? Can we compare something like a “coordination index” of one application with another?

3 Coordination Models

When coordination theory is applied to computer science, the key issue is *managing dependencies* among activities. To formalize and better describe these interdependencies it is necessary to separate the two essential parts of a parallel application namely, *computation* and *coordination*. This sharp distinction is also the key idea of the famous paper of Gelernter and Carriero [16] where the authors propose a strict separation of these two concepts. The main idea is to identify the computation and coordination parts of a distributed application. Because these two parts usually interfere with each other, the semantics of distributed applications is difficult to understand.

To fulfill typical coordination tasks a general coordination model for computer science has to be composed of four components (see also [33]):

1. Coordination entities as the processes or agents running in parallel which are subject of coordination,
2. A coordination medium: the actual space where coordination takes place,
3. Coordination laws to specify interdependencies between the active entities and
4. A set of coordination tools.

If we take a close look at the UNIX operating system, we can say that it already implements such a model. The coordination entities are the usual UNIX processes, and there are numerous coordination media: files, pipes, signals, messages, the coordination laws are determined by the semantics of the UNIX system calls, and finally the coordination tools are shells, X11 window interface and the like.

From Coordination Models to Coordination Languages

In [16] the authors state that a coordination language is orthogonal to a computation language and the “*linguistic embodiment of a coordination model*”. Linguistic embodiment means that the language must provide language constructs either in form of library calls or in form of language extensions as a means to materialize the coordination model. Orthogonal to a computation language means that a coordination language extends a given computation language with additional functionalities which facilitate the implementation of distributed applications.

One key issue in realizing such a coordination language is the question which kind of underlying communication model should be used to implement it [4]. As a means to achieve coordination some sort of communication has to be established between active entities. Note, that popular communication models such as message passing, data flow, or shared memory models can already be regarded as models for coordination. However, distributed applications require cooperation scenario such as client/server or task farming models which are concepts distinct from (basic) communication models.

We distinguish two different fundamental approaches for communication metaphors: coupled communication and uncoupled communication.

Coupled Communication versus Uncoupled Communication

Using communication as a method to achieve coordination implies some exchange of information. This exchange can either be realized by means of specific message passing routines (coupled communication), like in [28, 56] or by the introduction of a blackboard system (uncoupled communication) where information will be stored and hence made visible not only for a specific destinee, but for other entities too. Information becomes “public”, compared to message passing systems (where it is “private”), and all active entities which have access to this blackboard system can eventually read this information. The exchange of information is called “uncoupled” because the sender (of information) does not have to specify a particular receiver of the message, and more importantly, sender and receiver are unrelated to each other.

Uncoupled communication [25] has several advantages compared to message passing systems. First of all, the information becomes a first class entity, therefore an object to reason about. Furthermore information can become persistent, allowing several agents to consume the information even if the originator of the messages does not exist anymore. The different characteristics are summarized in Table 2.

Primitive	Uncoupled Communication	Coupled Communication
Send	Destination not specified, anonymous	Broadcast to group or to specific destinee
Receive	Sender anonymous	Active, sender can be anonymous

Table 2: Classification of uncoupled and coupled communication.

The strict separation of concern has led to the development of several coordination models and corresponding languages, with Linda [25] as the most prominent representative of so called tuple space models. Other models are based on message passing paradigms, object-oriented techniques, or multi-set rewriting schemes.

Linda

Linda [15] is based on the concept of uncoupled communication. Synchronization and communication in Linda is performed via a tuple space, e.g., an associative shared memory pool in which data in form of so called *tuples* is added to, removed from, and read from concurrently. A tuple consists of one or more fields. The pool is associative because data is accessed by referring to data types of the tuple fields. Every communication is handled via this tuple space and there is no point to point communication anymore. Processes are decoupled in time and space because they do not have to execute at the same time (as for message passing systems), and they do not have to know each others identifiers.

The communication metaphor in Linda is also called *generative* communication because it does not rely on message passing nor on a shared variable approach, but on the generation and consumption of tuples in the tuple space. A data producing process generates a new data object and puts it into the tuple space. The tuple space itself is a multi set of tuples (and not a set), that is, two identical tuples exist as two separate copies in the tuple space and can hence be retrieved by different processes.

Process handling is done by the introduction of *active tuples* as opposed to *passive tuples* (data). Active tuples are tuples with associated processes which are executed concurrently to their generator processes. Active tuples turn into passive ones upon completion of their processes. Values of these new passive tuples are the return values of the computations. Processes accessing an active tuple which has not yet carried out its computation are blocked until the data is available.

In order to access a tuple in a tuple space Linda uses *templates*. A template describes the expected layout of a tuple in the tuple space, its data types and consists of either a *formal* or an *actual*. A *formal* is a placeholder for a given value, a name of a variable of the host language where Linda is embedded, whereas an *actual* is the value of a given type.

Although the Linda model is quite powerful it holds some important problems.

- *Update operations* on tuples are difficult to express, this must be a sequence of atomically executed `in` and `out` operations.
- *Modularity*: a single global multi space circumvents modular data structures and violates data privacy and modularity. To overcome this problem multiple tuple spaces have been introduced relatively early [26], [31]. Data hiding aspects are addressed in [43].
- *Semantics*: the semantics of Linda's primitives have never been defined explicitly, but vary from implementation to implementation. For example, the `eval` operation executes for each supplied formal an independent thread of execution in some implementations, whereas in some other implementations all arguments are grouped together and executed sequentially in one single process. This might lead to deadlocks under certain circumstances. There are several propositions to define a clean semantics for a coordination language, see [21] as an example.
- *Performance problems*: the implementation of a centralized tuple space (as for example in [52] with its tuple server) may lead to performance problems. However, these problems can be avoided with a distributed implementation of a global tuple space in order to preserve the advantages of the Linda model by simultaneously achieving almost the performance of message passing programs [10], [46].

The Linda model has been inspiration for various research projects, and the following section will give a short summary of these projects, without claim of completeness.

Linda and her Friends

Laura [57] uses the tuple space abstraction as a tool for coordination of services, in Objective Linda [32], the tuple space abstraction turns into an *object space* to match entire objects, whereas in Piranha [14] Linda's tuple space is used for networked based load balancing functionality. The PageSpace [22] effort extends Linda's tuple space onto the World-Wide-Web and BONITA [51] addresses performance issues for the implementation of Linda's `in` and `out` primitives.

Bauhaus Linda [17] is a Linda derivate, which unifies tuples and tuple space to a single data structure: multi sets (mset). Therefore tuples are matched by the set inclusion operation and not by Linda's usual order and type-sensitive matching scheme. Also the basic operations, `rd` and `in` now return multi sets and have no side effects to their formal parameters.

The MANIFOLD [6] coordination model is based on the Idealized Worker Idealized Manager model [5] and focuses on separation of computation concerns from communication concerns by introducing a descriptive language to coordinate distributed applications. The language is event driven and uses a dynamic connection graph to coordinate active entities. It differs fundamentally from Linda and its derivate because it does not use a shared medium for coordination. A development close to the MANIFOLD approach has recently been presented in [30].

The Actor [2] has been extended to ActorSpaces [1] in which the original point to point communication of Actors is replaced by pattern oriented broadcasting.

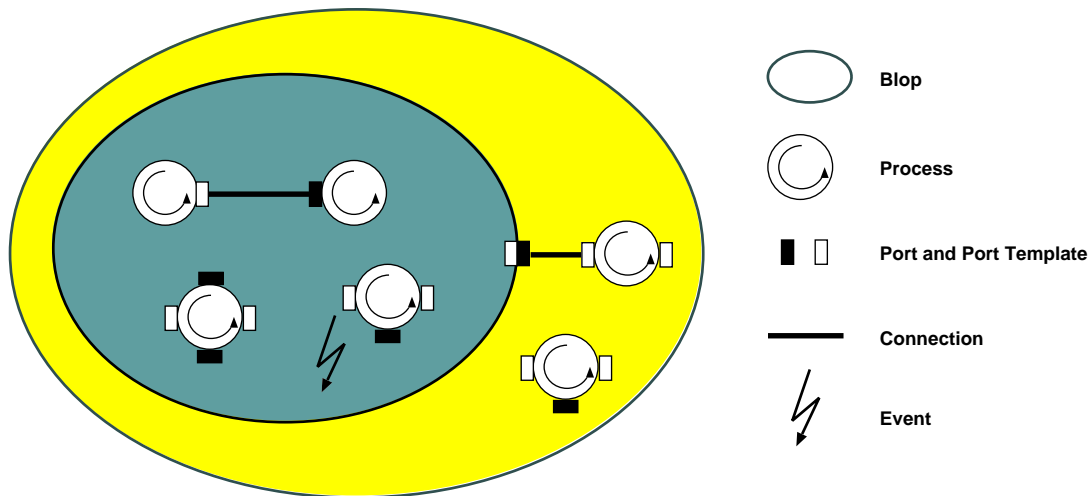
Multi-set rewriting techniques for coordination are introduced in Gamma [7].

Issues for concurrent agent-oriented programming are addressed in Interaction Abstract Machines [3] which are based on Linear Objects [11] a model which describes the interaction of agents in Linear Logic.

4 The Coordination Model of STL

The coordination model of STL is composed of five elements:

1. Processes, as a representation of active entities;
2. Blops, as an abstraction and modularization mechanism for processes and ports;



3. Ports, as the interface of processes/blops to the external world;
4. Events, a mechanism to react to dynamic state changes within a blop;
5. Connections as a representation of connected ports.

According to the general characteristics of what makes up a coordination model and corresponding coordination language, these elements are classified in the following way.

- The *Coordination Entities* of STL are the processes (implemented as threads) of the distributed application.
- The *Coordination Media* of STL are the following: events and connections serve as the medium which enables coordination, and a blop is the medium in which coordination takes place.
- The *Coordination Laws* are defined through the semantics of the *Coordination Tools*. STL's coordination tools are the operations defined in the computation language which work on the port abstraction. The coordination laws are defined through the semantics of these tools and the semantics of the interactions with the coordination media.

Figure 4 gives a first intuitive definition of the programming metaphor on which STL is based. An STL application consists of a hierarchy of *blops*¹ in which several *processes* run. Processes communicate and coordinate themselves via *events* and *connections*. *Ports* serve as the communication endpoints for connections which result in pairs of matched ports.

Processes have sensors called *ports* which stick out of the process interior and are visible inside a blop or through several blop hierarchies. These sensors are either defined statically (at compile time) or created dynamically (at runtime). Dynamic creation of ports is realized in the computation language by producing and consuming port signatures inside a blop.

Ports are used to establish a *Connection*, that is a private communication medium between two or more processes. A match relation defined on port signatures determines the set of ports which form a connection. Connections have different types, according to the characteristics of the matched ports. A process may dynamically produce a port signature in a blop and another one may consume it by specifying a port template, analogous to the data templates known from the Linda coordination model.

An *Event* inside a blop is a mechanism to dynamically react to changes of a STL program. Events are raised locally (local to a blop), their effect is limited to the blop in which they have been raised. They are

¹We used circles to reason about programs written in STL and after a while we had to name these circles somehow, for one reason or another the shape of these circles reminded us of the shape that bubbles produce when they come up of a viscous liquid. This is called "Blub" or "Blop" in German.

used to dynamically create processes and/or blops. To handle events, STL provides an event definition block inside a blop which is used to define appropriate event handling routines for certain events. If such a routine is not defined, that is, the blop is not tuned to react to it, the event simply vanishes.

4.1 Blops

A blop is an abstraction for an agglomeration of objects to be coordinated and serves as a separate name space for port objects, processes, and subordinated blops as well as an encapsulation mechanism for events. It is therefore the medium, the Coordination Space [34] of STL in which coordination takes place.

Blops have the same interface as Processes, i.e., a name and a possibly empty set of static ports, and can be hierarchically structured. We distinguish the declaration of a blop from its instantiation, with the exception of the default meta-blop, called `world`. Implicitly instantiated by the system, this blop serves as the basic environment in which every other activity is embedded, i.e., an STL application runs per default in this meta blop `world`.

The name of a blop is used to create instances of a blop object. The creation of a blop is a complex recursive procedure, it includes the initialization of all static processes and ports defined for this blop and subordinated blops.

In the current implementation blops are distributed onto the hosts of the parallel machine. Figure 3 shows the definition of the two blops of Figure 4 in STL syntax, the port definitions will be explained later.

```
blop world {
  ...
  blop sieve(a, b) {
    subscribe Porttype a<"CONNECTOR">; // Declaration of blop sieve
    publish   Porttype b<"CONNECTOR">; // Two ports a, b
    // Each port has a type
    // and name
  }
  ...
  create blop sieve s(); // Create blop sieve s
}
```

Figure 3: Blop Declaration and Invocation in STL.

4.2 Processes

STL knows only one type of active entity, called a process. A process in STL is a typed object, it has a name and a (possibly empty) set of static ports. As for blops, the handling of processes in STL is done in two steps: (1) declaration of a process type, and (2) instantiation and invocation of such a declared process. However, processes can generate dynamic ports during their lifetime, see below.

Processes in STL do not know any kind of process identification, instead a black box process model is used: a process runs with a set of ports; it does not have to care about from which process information will be transmitted to or received from.

Activation Policy, Process Control

Processes can only be activated in the coordination language. This is done through the instantiation of a process object inside a blop. To dynamically create new processes the process object instantiation must be done in the body of an event, so that the dynamic creation of new processes is the result of an event inside a blop.

Figure 4 shows an example of an STL process type `worker` with two static ports `in` and `result`, the syntax and semantics of the port definition will be explained in the next section.

```

process worker(in, result) {           // Process type worker
  subscribe PVMPort in<"WORK">       // in port
  publish  PVMPort result<"RESULT">  // result port
  func worker;                       // Implementation
}
create process worker w;              // An instance of this process type

```

Figure 4: Process Declaration and Invocation in STL.

Process termination is implicit: once the function which implements the process inside the computation language has terminated, the process disappears from the blop. The programmer can specify what should be done with the process' ports upon termination. Either the ports remain in the blop, so that later on processes can connect to them and retrieve data from them, or the ports may disappear with the process. The former case supports the uncoupled communication metaphor, processes are uncoupled in time through the port abstraction. It also supports a certain degree of security, because the information is not available in the whole blop but only in specific ports.

4.3 Ports

Ports are an abstraction which is used by processes and blops to establish connections to other processes/blops, i.e., every communication in STL is handled via a connection and therefore over ports. A port has a *name* and a set of well defined *attributes* which we call the port's *signature*.

The combination of port attributes results in a port type. We distinguish static and dynamic ports. A *static* port is an interface of a process or blop defined in the coordination language, whereas a *dynamic* port will be created dynamically at runtime in the computation language. However, the type of the dynamic port, i.e. its attributes must be defined in the coordination language.

Both static and dynamic ports are represented in a blop by port signatures. A process providing a port publishes it, another one subscribes to it, if both match, a connection will be established.

4.3.1 Port Attributes

Ports have several attributes which must be respected in order to establish a connection between them. The port attributes are the following:

- **Saturation.** Per default a port is totally bound if exactly one other port has matched to its signature. This can be overwritten with the `saturation` attribute of a port. We enable a specific number of other ports to connect to this port (`saturation=N, N> 0`), or unlimited number (`N == '*'`). With `saturation=*` the port is never bound.
- **Behavior after process termination.** Process' ports appear and disappear with the process they belong to. The `persistent` attribute changes this behavior, the port's signature remains in the blop upon termination of the process. Therefore the port can become a key to a connection, that is, a restricted repository for information items to which processes can connect to, even after the original data producing process has terminated.
- **Visibility.** Ports are objects whose visibility is per default limited to the blop where they are created. For nested blops it might be desirable to extend this visibility, and allow processes which are defined in the hierarchy "above" of the current blop to connect to these process' ports directly.
- **Semantics of flow of information.** Ports are a mechanism to store and transfer data. We provide the two classical communication paradigms: blackboard communication and stream communication. For blackboard communication the information can be retrieved from the port in an unordered sequence, and information can be retrieved more than once. As a consequence the operations in the computation language which work on the process' ports are defined so that information can

Attribute	Example	Explanation
Saturation	<code>saturation=2</code>	Allows exactly two other ports to connect to; default: 1
Behavior	<code>persistent</code>	Remains after process termination, default: disappears
Visibility	<code>visibility=global</code>	port is visible in all other blops, default: local
Communication	<code>blackboard, stream</code>	Semantics of flow of information
Capacity	<code>capacity=5</code>	Limits the capacity of a port to 5 data items; default: ∞
Synchronization	<code>synchron, asynchron</code>	Semantics of message passing model

Table 3: Attributes of a Port.

be removed or only read from the blackboard. Also a mechanism is provided to access selectively only a particular information on the blackboard.

For stream communication this selection scheme is not necessary, the information will be provided in the order it has been placed into the port. Therefore the sequence of information items which are forwarded through this port is predefined by the local arrival rate of the information at the port. It is important to note that the *local* arrival rate, as seen from the port's point of view will determine the message order and not a possibly global order.

- *Capacity.* Since ports can act as repositories for information, the `capacity` attribute determines the capacity of the port in terms of message items.
- *Type of message synchronization.* Synchronous as well as asynchronous message passing is supported. Synchronous communication is defined so that the data producing operation on the port blocks until the port gets bound.

Asynchronous means that no data producing operation on the port blocks the process, no matter in which state the port is.

In combination with the capacity of the port, a weaker synchronized model can be realized. Data producing operations block after the capacity of the port has been reached.

Table 3 gives an overview of the attributes of a port, combinations of attributes lead to port types, which will be explained in the next section.

4.3.2 Basic Port Types

The combination of different port attributes yields to different port types. We have identified three major port types: `PVMPort`, `CSPPort` and `TSPort`. Variants of these types are possible and can be defined by the user.

`PVMPort`:

The `PVMPort` can best be characterized as a PVM communication endpoint. Two matched `PVMPorts` result in a PVM connection with the following semantics: Every `out` operation on such a port is non blocking, the port has an infinite storage capacity, it is only locally visible, it matches to exactly one other port, disappears with the process and follows the stream communication model².

`CSPPort`:

The `CSPPort` differs from the `PVMPort` in 2 features. A `CSPPort` uses a synchronous message passing model, i.e., `out` operations on this port block until a corresponding communication partner has executed an `in` operation on it. This is realized by specifying a storage capacity of one message item. In analogy to Hoare's CSP model [28] we baptized the port `CSPPort`, because the resulting connection implements the CSP model of message passing. However, the difference to

²Note, that the original communication model in PVM differs from this semantics because messages are tagged with a message tag in PVM and therefore they can be received selectively, independent of the send order.

the original CSP model is that the data producing process does not have to know the recipient of the message, the communication is still anonymous but synchronous.

TSPort :

The TSPort stands for “Tuple Space Port”. As the name indicates, the resulting connection has a tuple space semantics à la Linda. In contrast to the previous port types this type allows processes to retrieve messages using a match relation defined on the signature of the message, i.e., in the order defined by the process and not by the arrival rate of messages in the connection.

By changing the behavior after process termination, the uncoupled communication metaphor is supported. Processes can connect to this blackboard via the port signature at any time if the port signature remains visible in the blob and survives the process. Other processes can connect to the port in the future and retrieve the information which is stored in the blackboard. This provides a certain degree of privacy and encapsulation for communicating processes which is not present in the original Linda model. In order to access the information, the process must specify both, the port signature (to get access to the blackboard) and the signature of the data item to retrieve. Modularity is supported in so far as the blackboard serves as a private name space for a restricted set of processes. Therefore each module can independently use the same message tuples without interfering with other modules.

Note a subtle difference to the original Linda model: processes do not belong to the tuple space with which they communicate, but are grouped around, outside of the blackboard.

Figure 5 shows the definition of the TSPort in STL syntax.

```
port TSPort {
  saturation=*;
  communication_metaphor=blackboard;
  visibility=local;
  capacity=inf;
  communication_sync=asynchron;
  behavior=persistent;
}
```

Figure 5: TSPort definition in STL.

4.3.3 Variations of the Basic Port Types

Combinations of these basic port types are possible, for example to define a $(1:n)$ PVM type of style connection, the saturation characteristics of an PVMPort can be augmented to n .

Another interesting variation is the $(1:n)$ CSPPort. The data producing process blocks until all n processes have connected to the port, and every out operation returns only after all n processes have received the data item.

4.3.4 Port Matching

The matching of ports is defined as a relation between port signatures.

Note, the match relation is not a static relation which can be determined at compile time, but depends on the current state of the port relative to its attributes. In other words, although the signature of two ports may match they do not match at runtime because, e.g., the amount of communication partners which are allowed to connect to this port is limited (through the `saturation` attribute). We call this the (dynamic) precondition under which matching is possible. We identified three important qualities which a port must fulfill in order to match to another port: (1) both ports must not be saturated, (2) both belong to the same level of abstraction, i.e., are visible within the same hierarchy of blobs, and (3) both belong to different objects (either process or blob).

In the following we start with a general discussion of what classes of matching are possible and conclude with the presentation of the class implemented in STL.

We define the set of port types \mathcal{PT} ; the set of port names as \mathcal{PN} , the port's signature as a tuple: $[pt : pn]$, with $pt \cup \{*\} \in \mathcal{PT}, pn \cup \{*\} \in \mathcal{PN}$, $*$, denotes the wildcard operator. The set of all port signatures currently used at time t is defined as $\mathcal{P}_S(t)$. The potential match partners of a given port's signature $[pt : pn]$ is dependent on the specification of the port to get matched.

Supplied	Set of matched signatures	Abbreviation
$[pt : pn]$	$\{[pt : pn]\} \in \mathcal{P}_S(t)$	MEx
$[pt : *]$	$\{pn \in \mathcal{PN} \setminus \{*\} \mid [pt : pn]\} \subseteq \mathcal{P}_S(t)$	MPTE _x
$[* : pn]$	$\{pt \in \mathcal{PT} \setminus \{*\} \mid [pt : pn]\} \subseteq \mathcal{P}_S(t)$	MPE _x
$[* : *]$	$\mathcal{P}_S(t)$	Mall

Table 4: Potential match partners for a given port signature

We distinguish four different classes, see Table 4. MEx denotes exact matching, both elements $[pt : pn]$ of the port's signature must match exactly. MPTE_x says that only the port type part of the port's signature must match exactly, and for MPE_x the port name part must fit exactly. Mall is the wildcard match specification, it matches with all other port's signatures currently available in a blop or a hierarchy of blops.

From the four alternatives described in Table 4, the MEx variant is used in STL. Only port signatures of the same type match. This is reasonable because the combination of matched ports determines the characteristics of the resulted connection. It is hard to define the semantics of a connection built through an `PVMPort` and `TSPort`. As a consequence no wildcard operators are supported, the publication and subscription of ports must be done with fully specified port signatures.

4.3.5 Static Ports

The publication and subscription of static ports is done automatically upon start-up of a process or blop. Consequently all static ports get eventually matched. Seen from this point of view, a blop performs a certain activity, upon creation of a new process or blop it matches as many static ports as possible.

Recall Figure 4 where process `worker` has two static ports of type `PVMPort`, `in` and `result`. These ports match with corresponding ports of other processes having the *port name* "WORK", and "RESULT" and *port type* `PVMPort` as part of their signature. Their process local presentation (`in` and `result`) however might be different, it describes the object port as seen from the process' point of view, *port name* is part of the port's signature and used by the match operation inside the blop.

Static ports are either used for processes, and more importantly for blops. Blops therefore have the same input/output interface as processes so that the granularity of a program can be changed very easily. Instead of dynamically creating new processes, a new blop is created. Of course this depends on the implementation of the STL model, in our case, it is implemented with PT-PVM [35] as the underlying communication and process management platform, and we implemented blops as heavy-weight UNIX processes and STL processes as light-weight processes running in a blop.

Another reason for static ports is the definition of particular events at coordination level, especially the unbound event, see the following sections.

Port 1 (p_1)	Port 2 (p_2)	Match
<code>PVMPort in<"WORK"></code>	<code>PVMPort in<"WORK"></code>	yes
<code>PVMPort in<"WORK"></code>	<code>PVMPort in<"RESULT"></code>	no
<code>PVMPort in<"WORK"></code>	<code>TSPort in<"WORK"></code>	no

Table 5: Examples of matching/non matching Ports.

4.3.6 Dynamic Ports

As already stated dynamic ports will be created in the computation language, their type however must be specified in the coordination language.

For dynamic ports a `blop` serves as a container for port signatures. In contrast to static ports, dynamic ports are produced and consumed in a `blop`. A process producing a port puts its signature into the `blop`, a process connecting to it consumes this signature in order to establish a connection.

By performing subscribe and publish operations, processes actively set the time when matching occurs and therefore when a connection gets established.

```
process master(...) {
  dynamic PVMPort ResultPort;
  func master;
}
```

Figure 6: Dynamic Port in STL.

Figure 6 shows an example of a dynamic port in STL. The process `master` creates at runtime new ports of type `ResultPort` which have the characteristics of a `PVMPort`. Inside the computation language new types `ResultPort` and `ResultPort_T` are used to instantiate new port objects. `ResultPort` is used for publish operations, it represents the actual port. `ResultPort_T` is the corresponding template type which is used for subscribe operations. Note, no local representation is used, only the type `ResultPort` is specified in the coordination language.

4.4 Connections

Connections between processes have either stream semantics or blackboard semantics.

4.4.1 Connections as Streams

Connections which serve as private communication stream between processes can have the following semantics. We follow the formal definition of channels in [5], however since our connections are per default bi-directional, only a subset of the channel semantics of [5] is provided.

- S channel: totally synchronous, no capacity to store information, built with two `CSPPort` ports;
- BB channel: disconnects *all* ends of the connection automatically when at least one process disconnects from it;
- KK channel: connection remains when one process disconnects, no matter how many processes disconnect from it.

4.4.2 Connections as Blackboards

Based on the `TSPort` port type, a connection with Blackboard semantics is established. The capacity of the Blackboard is determined by the `capacity` property of the port type.

4.5 Events

Events can be triggered using a condition operation on a port or by a process directly using the `raise()` operation in combination with the event's name. The event is handled by an event handler inside the `blop`.

An event declaration may be attached to a condition which determines when the event will be executed in the `blop`. The conditions are related to ports of processes or `blops`. Table 6 gives an overview of conditions on ports which trigger an event. Whether an event must be triggered or not will be checked

by the system every time the port is used by a process, otherwise a condition like `isempty` would uninterruptedly trigger events for ports of processes which are about to get initialized.

After an event has been triggered, a blop is not tuned anymore to handle subsequent events of the same type. In order to handle these events again, the event handling routine must be re-installed which is usually done in the event handling routine of the event currently processed.

Condition	Explanation
<code>accessed</code>	Port has been accessed
<code>unbound</code>	Port has no communication partners
<code>isempty</code>	Port contains no data
<code>isfull</code>	Port is full
<code>msg_handled(int n)</code>	Port has handled n messages
<code>less_msg_handled(int n)</code>	Port has handled up to n msg.

Table 6: Conditions on ports.

Very useful is the `unbound` condition on ports. It makes it possible to construct parallel software pipelines very elegantly. If we reconsider Figure 4 and extend it to Figure 7, we see the interaction of event conditions and ports in STL. Firstly an event `new_worker` is declared. The event is attached to a `unbound` condition of the `out` port of the initial process `w`. If this process either reads or writes data from/to the port, the event `new_worker` is triggered because there are no other ports to which `w.out` is currently bound, so `unbound(w.out)` is `TRUE`. The event body of the event declaration of `new_worker` creates a new process of type `worker`, the blop matches now the two ports (`w.out` with `new.in`) and the information can be transferred from `w` to `new`. The same mechanism recursively works for the `out` port of the created process `new`, because a condition is attached on its port.

However events can also unconditionally be triggered from the computation part of the application by using the `raise()` primitive.

```

event new_worker() {
  create process worker new;
  when unbound(new.out) then new_worker();
}
process worker(in, out) {
  subscribe MyPort in<"WORK"> // in port
  publish MyPort out<"WORK"> // out port
  func worker; // Implementation
}
when unbound(w.out) then new_worker(); // Attach event to port
create process worker w; // An instance of this process type

```

Figure 7: Event Handling in STL.

4.6 Primitives

We can now give an overview of the primitives used in the computation language.

4.6.1 Operations on dynamic Ports

`<porttype>`:

A `porttype` corresponds to a port type definition in the coordination language, `<porttype>` is the class of all port types in STL.

`<porttype_T>`:

A `porttype_T` denotes the template type used to subscribe to a port of type `porttype`. A port with type `porttype_T` gets bound with a `inport` or `subscribe` operation.

`void publish(char *name, <porttype> p):`
 Places the signature of port `p` in the current blop, `<porttype>` must be a defined port type, the operation is non blocking. In order to provide a symmetric interface, we provide a `outport` operation with the same semantics.

`void subscribe(char *name, <porttype.T> p):`
 Reads a signature which corresponds to a port of type `porttype` from the blop and binds port `p` to the connection which is represented by this signature. `subscribe` blocks the calling process in case the port signature is not present in the blop.

`void inport(char *name, <porttype.T> p):`
 Analogous to `subscribe`, with the difference that the port signature will be removed from the blop, consequently the port represented by this signature can not be accessed by other processes. As for `subscribe`, `inport` blocks the calling process in case the port signature is not present in the blop.

4.6.2 Operations to transfer Data

`<basic_data_type>:`
 A `<basic_data_type>` (BDT) is the class of basic data types in STL. It is one of the following data types: `IntObject`, `DoubleObject`, `CharObject`, `ByteObject` and is used to compose a message of type `Msg`. Each type can be supplied with an additional size parameter to specify the amount of objects. For example `CharObject<640>` defines a character buffer consisting of 640 characters.

`<basic_data_type_template>:`
 Analogous to the BDT, `<basic_data_type_template>` (BDTT) denotes the class of types to receive a particular data item in a message. BDTT is one of the following: `IntTempl`, `DoubleTempl`, `CharTempl`, `ByteTempl`. The size parameter is used accordingly.

`Msg m(DT o, DT o1, DT o2, ...):`
 Creates a message which is composed of one or more data types. In case the message is used in a blackboard connection, it will be accessible in a Linda like fashion. To consume it from the blackboard, the message layout must match with an appropriate message on the blackboard, and therefore `DT` is either a BDTT to specify the type of data to receive, or a BDT to address a specific message on the blackboard.
 For stream connection data can only be received, i.e., only objects of type BDT can be used to compose the message.

`void in(PORT p, Msg m):`
 Retrieves `Msg m` from the connection which is connected to port `p`. Always blocks the process until either the requested data item is present (for blackboards), or the specified data item is available (for streams). `PORT` denotes the class of all defined port types.

`void rd(PORT p, Msg m):`
 Like `in` but does not remove the data item from the connection. Particularly used if the connection has blackboard semantics.

`void out(PORT p, Msg m):`
 Places `m` in a connection which is connected to port `p`. Whether the operation blocks or not depends on the port type.

4.6.3 Operations on Events

`void raise(char *eventname):`
 Triggers unconditionally the event `eventname` in a blop.

As far as the implementation of these primitives is concerned, we use PT-PVM [35] a software platform for programming multi-threaded applications on a cluster of workstations as the underlying com-

munication and process management platform. PT-PVM is part of the CoLMA³ effort of the University of Fribourg, which aims at developing tools for coordination of multi-threaded applications on a cluster of workstations.

5 Our Autonomy-based Multi-Agent System in STL

5.1 Introduction

In order to illustrate the expressiveness and appropriateness of STL, we give as example a preliminary distributed implementation of our autonomy-based multi-agent system, wherein few simplifications have been brought in comparison with the serial implementation ([20]). We illustrate with a collective robotics simulation, where agents seek for objects spread in their environment.

The environment in which agents move is a torus grid with a four connectivity (each cell has four neighbours). Agents comply rigorously with the model introduced (see Figure 1). They sense the environment through their sensors and act upon their perception at once.

The environment is split into sub-environments, each of which being handled by a blop, as indicated on figure 8, thus providing an independent functioning between sub-environments. Note that blops have to be arranged in accordance with the topology of the environment they implement.

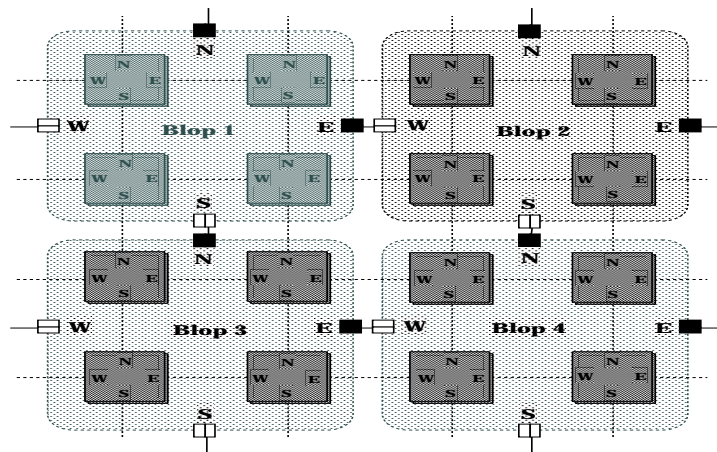


Figure 8: Splitting an environment made up of cells into four blops

5.2 Global Structure

By default, every STL implementation runs in STL's meta-blop *world* which is composed in this case of an *init* process, responsible for the global initialization of the system, and a set of pre-defined blops, (called *se*), each one handling a sub-environment.

The *init* process has two static ports (of type `PVMPort`) for every blop to be initialized (Figure 9 illustrates the connections between the *init* process and a blop *se*). The rôle of the *init* process is twofold: first, to initialize the agents (number of agents, characteristics of every agent) within every blop through its *cre_Agts* port; secondly, to set up the sub-environment (size, number of objects) of every blop through its *cre_SubEnv* port.

³Coordination Language for Multi-threaded Applications.

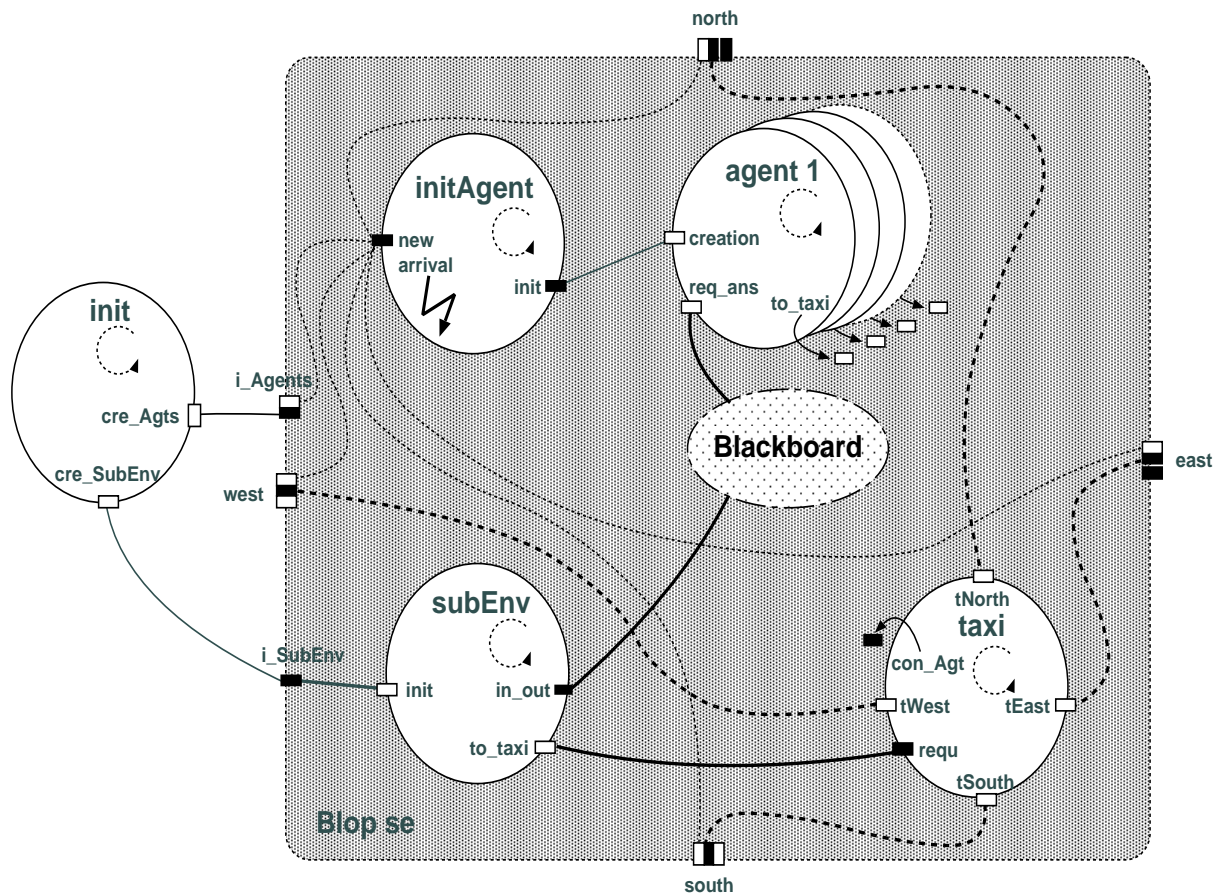


Figure 9: *init* process and *blop se*: solid and dotted lines are introduced just for a purpose of visualization

5.2.1 Blop se

Figure 9 shows the basic organization of processes within a *blop se* and their coordination through ports. Two types of processes may be distinguished: processes that are part of the coordination platform, namely *initAgent* and *taxi*, and processes that are intrinsic to the application, viz. *subEnv* and *agent* processes.

5.2.2 Ports of a Blop

Each *blop* has four static communication ports of type `PVMPort` (named *north*, *south*, *west*, *east*) referred to as *direction* ports, which are used for agent migration: agents roam indeed in the whole environment. For the time being, the topology between *blops* is set in a static manner, by performing *publish/subscribe* operations with appropriate strings given at creation time and representing the desired topology, like for instance the one presented on figure 8. The four ports of a *blop* match as well with its two inner processes *taxi* and *initAgent*.

5.2.3 *initAgent* Process, *newAgentEvt* Event

The *initAgent* process is responsible for the creation and the initialization of the agents. It has two static ports: *newArrival* and *init*. The *newArrival* port is connected to all *direction* ports of the *blop* within which

it resides. As soon as a value comes to this port, the *initAgent* process copies it onto its *init* port. In the meantime, the *newAgentEvt* event is triggered and it will create a new *agent* process, which through its *creation* port will read the value that was previously written on the *init* port of the *initAgent* process. Values that are transmitted feature for instance the *state* of the agent to create.

Note that the language binding for all code examples (starting with figure 11) is somewhat different as introduced in section 4.6 because we used the C++ version of the primitives for STL. Therefore procedure calls turn into method invocations.

```

process initAgent(newArrival, init) {
  publish PVMPort newArrival <"NEW-ARRIVAL">;
  publish PVMPort init <"AGENT-INIT">;
}
event newAgentEvt {
  create process agent a;
}
create process initAgent i;
when accessed(i.newArrival) then newAgentEvt;

```

Figure 10: Declaration and Invocation of process *initAgent* and event *newArrival* in STL.

```

void initAgent(PVMPort newArrival, init) {
  ByteTempl<32> state;
  Msg stateTp(state);
  while (TRUE) {
    newArrival.in(stateTp);
    init.out(stateTp);
  }
}

```

Figure 11: Implementation of the process *initAgent* in C++.

5.2.4 agent Process

This process has two static ports (*req_ans* of type `TSPort` and *creation* of type `PVMPort`) plus *to_taxi* a dynamic port. As already stated, this process reads on its *creation* port some values (its *state*). All *req_ans* ports of the agents are connected to a *Blackboard*, through which agents will sense their environment (*perception*) and act into it (*actions*), by performing Linda-like *out/in* operations with appropriate messages. The type of action depends on the type of *control Algorithm* implemented within the agent (see the architecture of an agent on figure 1). The *to_taxi* port is used to communicate dynamically with the *taxi* process in case of migration: the *state* of the agent is indeed copied to the *taxi* process. The decision of migrating is always taken by the *subEnv* process.

5.2.5 subEnv Process

The *subEnv* process handles the access to the sub-environment and is in charge of keeping data consistency. It is also responsible for migrating agents, whose next decided action will make them crossing the border of a sub-environment. It has a static *in_out* port (of type `TSPort`) connected to the *Blackboard* and a static port *to_taxi* connected to the *taxi process*. Once initialized through its *init* port, the *subEnv* process builds the sub-environment. By performing Linda-like *in/out* operations with appropriate tuples, the *subEnv* process will process the request of the agents (e.g. number of objects on a given cell, move to next cell) and reply to their request (e.g. *x* objects on a given cell, move registered). When the move of an agent will lead to cross the border (cell located in another blob), the *subEnv* process will first inform the agent it has to migrate and then inform the *taxi* process an agent has to be migrated (the name of the agent and the direction the agent has to take will be transmitted).

5.2.6 The taxi Process

The *taxi* process is responsible for migrating agents across blobs. It has four static *direction* ports, which are connected to the four *direction* ports of the blob within which it stands. When this process receives on its static port *requ* the name of an agent and the direction towards where this agent has to migrate, it will create a dynamic `PVMPort` port *con_Agt* in order to establish with the appropriate *agent* process a communication, by means of which it will collect all the useful information of the agent (*state*). These values will then be written on the port corresponding to the direction to take and will be transferred to

```

void agent(TSPort req_ans, PVMPort to_taxi, creation) {
    ByteTempl<32> state, answer;           // Template
    ByteObject<32> *req;
    Msg stateTp(state);                   // Message
    boolean noMigration = TRUE;

    creation.in(stateTp);                  // Initialize with state

    while (noMigration) {
        req = make_preception_req();       // PERCEPTION
        Msg requestTp("request", req->id, *req);
        req_ans.out(requestTp);            // Put preception request
        Msg answerTp("answer", req->id, answer);
        req_ans.in(answerTp);              // Get answer

        req = control(answer)              // CONTROL Algorithm

        req = make_action_req();           // ACTION
        Msg requestTp("request", req->id, *req);
        req_ans.out(requestTp);            // Put request on Blackboard
        Msg answerTp("answer", req->id, answer);
        req_ans.in(answerTp);              // Get answer of requested action
        state = update_state(answer);       // Update internal state
        noMigration = migrate_p(answer);    // Check whether to migrate or not
    }
    PVMPort to_taxi;                       // Migration
    to_taxi.publish("taxi");                // Publish to_taxi port
    to_taxi.out(stateTp);                  // Transfer state to taxi process
    exit(0);
}

```

Figure 12: Implementation of the process *agent* in C++.

the *newArrival* port of the *initAgent* process of the concerned blop inducing the dynamic creation of a new agent process in the blop, thus materializing the migration.

6 Conclusion

In this paper, we focused on a generic model for autonomy-based multi-agent systems and demonstrated its suitability by addressing numerous applications such as collective robotics ones. We pointed out the major problem entailed by serial implementations, namely the problem of preserving the temporality of every agent and hence its autonomy. We sketched out a preliminary distributed implementation of an autonomy-based multi-agent system based on a first prototype embedding the coordination model of STL which itself is built on top of Pt-PVM, a platform [35] which provides high level flexible coordination patterns.

Eventhough STL is inspired by Linda and Manifold, it however differs in several points. Major differences between STL and Linda are: (1) STL uses a hierarchical multiple coordination space model, in contrast to the single flat tuple space of Linda. (2) In STL, port signatures get matched and the matching of data items is handled in several subenvironments. (3) STL uses a different process management (no *eval* operation). Differences between STL and Manifold are the following: (1) Connections in STL have two different semantics: either stream or blackboard. (2) The coordinator process which is essential in Manifold is not needed in STL, and (3) connections get established through a match relation defined on port signatures and not through an additional coordinator process. (4) STL uses a structured coordination space and not a flat one as Manifold does.

The development of our system based on STL can be seen as a starting point to benefit from its concepts that seem appropriate for multi-agent programming, among which are: 1) the absence of a central coordinator process, which does not relate to any type of entity in the multi-agent system; 2) the

```

void subEnv(PVMPort init, TSPort in_out, PVMPort to_taxi) {
    IntTempl id, nbrOfAgents, nbrOfObjects, nbrOfCells;
    ByteTempl<32> req, *resp;
    SubEnv *subenv;

    Msg initTp(nbrOfAgents, nbrOfObjects, nbrOfCells);
    init.in(initTp);

    subenv = init_env(nbrOfAgents, nbrOfObjects, nbrOfCells); // Build the sub-environment

    while(TRUE) {
        // Request - Answer loop
        Msg requestTp("request", id, req);
        in_out.in(requestTp); // Get request from Blackboard
        resp = decide_response(req, subenv);
        Msg answerTp("answer", id, *resp);
        in_out.out(answerTp); // Put answer on Blackboard
        if migrateP(resp) { // Agents migrates
            Msg migTp(id, CharObject<4>(getDirection(req)));
            to_taxi.out(migTp); // Inform taxi process
        }
    }
}

```

Figure 13: Implementation of the process *subEnv* in C++.

notion of ports avoiding any additional coordinator process; and 3) in despite of 2) the notion of blop hierarchy which in our case allow us to represent the encapsulation of the environment and the agent into a universe. In this respect STL is a first attempt to implement autonomy-based multi-agent systems. However, we are aware of the fact that the model as presented in this paper needs some improvements to better fulfil the needs for distributed multi-agent programming.

There are two major outcomes to this work. First, as autonomy-based multi-agent systems are aimed at addressing problems which are naturally distributed, our coordination platform provides a user the possibility to use an actual distributed implementation and therefore to benefit from the numerous advantages of distributed systems (in particular the implementation closer to the reality it simulates), making therefore this work a step forward in the *Autonomous Agents* community. Secondly, as the generic patterns of coordination for autonomy-based multi-agent implementations are embedded within the platform, a user can quite easily develop new applications (e.g. by changing the type of autonomy of the agents, the type of environment), insofar they comply with the generic multi-agent model.

Future works are twofold: first, a graphical user interface is planed to be developed in order to help implementing an application. Secondly, the STL coordination model is still to be extended in order to encompass as many as possible generic patterns of coordination, yielding in STL skeletons at disposal for general purpose implementations. Finally we also foresee to develop an STL-based coordination platform especially dedicated to autonomy-based multi-agent systems where we will stress on the most critical constraints for distributed implementations, that is, the preservation of every agent's temporality by dismissing coordination mechanisms exclusively embedded for purpose of implementation.

References

- [1] G. Agha and C. J. Callsen. ActorSpace: An Open Distributed Programming Paradigm. In *SIGPLAN Symposium on Principles & Practice of Parallel Programming PPOP*, pages 23–32, San Diego, California, May 19-22 1993. ACM Press.
- [2] G. Agha, S. Folund WooYoung, and Kim Rajendra Panwar. Abstraction and Modularity Mechanisms for Concurrent Computing. *IEEE Parallel & Distributed Technology*, 1(2):3–14, May 1993.
- [3] J.M. Andreoli, P. Ciancarini, and R. Pareschi. Interaction Abstract Machines. In P. Wegner G. Agha and A. Yonezawa, editors, *Research Directions in Concurrent Object Oriented Programming*. MIT Press, Cambridge Mass., 1993.

```

void taxi(PVMPort tNorth, tSud, tWest, tEast, requ, con_Agt) {
    CharTempl<4> direction;
    IntTempl id;
    ByteTempl<32> state;
    Msg stateTp(state);
    Msg init(id, direction);

    while(TRUE){
        requ.in(init);                // Initialization from subEnv

        PVMPort con_Agt;              // Consume the port which has been
        con_Agt.inport("taxi");       // published by agent
        con_Agt.in(stateTp);          // Get agent's state

        switch (direction) {          // Migration to other blob
            case "N":                 // Handle directions
                tNorth.out(stateTp);
                break;
            case "S":
                ...
        }
    }
}

```

Figure 14: Implementation of the process *taxi* in C++.

- [4] F. Arbab. Coordination of Massively Concurrent Activities. Technical report, CWI, Computer Science Department, Amsterdam, The Netherlands, 1995. CS-R9565.
- [5] F. Arbab. The Influence of Coordination on Program Structure. In R. H. Sprague Jr., editor, *Proceedings of the 30th Hawaii International Conference on System Sciences*, volume 1, pages 300–309, Wailea, Hawaii, 1997. IEEE.
- [6] F. Arbab, I. Herman, and P. Spilling. An Overview of Manifold and its Implementation. *Concurrency: Practice and Experience*, 5(1):23–70, February 1993.
- [7] Jean-Pierre Banâtre and Daniel Le Métayer. Programming by Multiset Transformation. *Communications of the ACM*, 36(1):98–111, 1993.
- [8] R.D. Beer. A dynamical systems perspective on agent-environment interaction. *Artificial Intelligence*, 72:173–215, 1995.
- [9] G. Beslon, F. Biennier, and J. Favrel. A Flexible Conveying System Based on Autonomous Vehicles. In *Proceedings of CARs and FoF'95*, volume 1, pages 115–120, Pereira, Colombia, August 1995.
- [10] R. Bjornson, N. Carriero, D. Gelernter, T. Mattson, D. Kaminsky, and A. Sherman. Experience with Linda. Technical report, Yale University, Department of Computer Science, August 1991.
- [11] M. Bourgois, J.M. Andreoli, and R. Pareschi. Extending Objects with Rules, Composition and Concurrency: the LO Experience. Technical report, European Computer Industry Research Centre, Munich, Germany, 1992.
- [12] R.A. Brooks. Intelligence without Reason. In *Proceedings of IJCAI-91*, Sydney, Australia, 1991.
- [13] R.A. Brooks. Intelligence without Representation. *Artificial Intelligence, Special Volume: Foundations of Artificial Intelligence*, 47(1-3), 1991.
- [14] N. Carriero, E. Freeman, D. Gelernter, and D. Kaminsky. Adaptive Parallelism and Piranha. *IEEE Computer*, 28(1), January 1995.
- [15] N. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, 1989.

- [16] N. Carriero and D. Gelernter. Coordination Languages and Their Significance. *Communications of the ACM*, 35(2):97–107, February 1992.
- [17] N. Carriero, D. Gelernter, and L. Zuck. Bauhaus Linda. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *Lecture Notes in Computer Science*, Berlin, 1995. Springer Verlag.
- [18] F. Chantemargue, T. Dagaëff, M. Schumacher, and B. Hirsbrunner. Coopération implicite et antagonisme. Technical Report 96-15, Computer Science Department, University of Fribourg, Fribourg, Switzerland, December 15 1996.
- [19] F. Chantemargue, T. Dagaëff, M. Schumacher, and B. Hirsbrunner. Coopération implicite et performance. In *Proceedings of the Sixth symposium on Cognitive Sciences (ARC)*, Villeneuve d’Ascq, France, December 10-12 1996.
- [20] F. Chantemargue, T. Dagaëff, M. Schumacher, and B. Hirsbrunner. The Emergence of Cooperation in a Multi-Agent System. Technical Report 96-16, Computer Science Department, University of Fribourg, Fribourg, Switzerland, December 22 1996.
- [21] P. Ciancarini, K. Jensen, and D. Yankelewich. On the Operational Semantics of a Coordination Language. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *Lecture Notes in Computer Science*, Berlin, 1995. Springer Verlag.
- [22] Paolo Ciancarini, Andreas Knoche, Robert Tolksdorf, and Fabio Vitali. PageSpace: An Architecture to Coordinate Distributed Applications on the Web. In *Proceedings Fifth International World Wide Web Conference*, volume 28 of *Computer Networks and ISDN Systems*, 1996.
- [23] T. Dagaëff, F. Chantemargue, and B. Hirsbrunner. Emergence-based Cooperation in a Multi-Agent System. In *Proceedings of the Second European Conference on Cognitive Science (ECCS’97)*, pages 91–96, Manchester, U.K., April 9-11 1997.
- [24] Keith S. Decker. Why study Coordination? <http://centaurus.cs.umass.edu:80/decker/latex/coord.dvi>, 1994.
- [25] David Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
- [26] David Gelernter. Multiple Tuple Spaces in Linda. In E. Odijk, M. Rem, and J. Syre, editors, *Proc. Conference on Parallel Architectures and Languages Europe (PARLE 89)*, volume 365 of *Lecture Notes in Computer Science*, pages 20–27, Berlin, 1989. Springer Verlag.
- [27] S. Harnad. The Symbol Grounding Problem. *Physica D*, 42, 1990.
- [28] C.A.R. Hoare. Communication Sequential Processes. *Communications of the ACM*, 21(8), August 1978.
- [29] J.H. Holland. Escaping brittleness: the possibilities of general purpose learning algorithms applied to parallel rule-based systems. In Michalski, Carbonell, and Mitchell, editors, *Machine learning, and artificial intelligence approach*, volume II of *Morgan Kaufmann*. Los Altos, CA, 1986.
- [30] A. A. Holzbacher. A Software Environment for Concurrent Coordinated Programming. In Paolo Ciancarini and Chris Hankin, editors, *First International Conference on Coordination Models, Languages and Applications*, number 1061 in LNCS. Springer Verlag, April 1996.
- [31] Keld K. Jensen. *Towards a Multiple Tuple Space Model*. PhD thesis, Institute for Electronic Systems, Department of Mathematics and Computer Science, Aalborg University, Aalborg, Denmark, 1994.

- [32] Thilo Kielmann. Designing a Coordination Model for Open Systems. In Paolo Ciancarini and Chris Hankin, editors, *First International Conference on Coordination Models, Languages and Applications*, number 1061 in LNCS. Springer Verlag, April 1996.
- [33] Thilo Kielmann and Guido Wirtz. Coordination Requirements for Open Distributed Systems. In *PARCO'95*. Elsevier, 1996.
- [34] O. Krone and M. Aguilar. Bridging the Gap: A Generic Distributed Hierarchical Coordination Model for Massively Parallel Systems. In *Proceedings of the '95 SIPAR-Workshop on Parallel and Distributed Computing*, Biel, Switzerland, October 1995.
- [35] Oliver Krone, B at Hirsbrunner, and Vaidy Sunderam. PT-PVM+: A Portable Platform for Multi-threaded Coordination Languages . *Calculateurs Parall els*, 8(2):167–182, 1996.
- [36] C. Langton, N. Minar, and R. Burkhart. The Swarm simulation System: a toolkit for building Multi-agent simulations. Technical report, Santa Fe Institute, 1996.
- [37] P. Lerena. Bio-machines. In *Artificial Life*, volume V, Nara, Japan, 1996.
- [38] P. Maes. Behavior-Based Artificial Intelligence. In *Proceedings of the Fifteenth Annual Meeting of the Cognitive Science Society*, pages 74–83, Hillsdale, NJ, 1993. Lawrence Erlbaum.
- [39] P. Maes. Modeling Adaptive Autonomous Agents. *Artificial Life*, 1(1 and 2), 1994.
- [40] T. W. Malone and K. Crowston. The Interdisciplinary Study of Coordination. *ACM Computing Surveys*, 26(1):87–119, March 1994.
- [41] H. Maturana and F.J. Varela. *Autopoiesis and Cognition: the realization of the living*. Reidel, Boston, MA, 1980.
- [42] O. Michel. Khepera Simulator version 1.0. User Manual. Technical report, University of Nice Sophia-Antipolis, Valbonne, France, 1995.
- [43] N. H. Minsky and J. Leichter. Law-Governed Linda as a Coordination Model. In *ECCOP Workshop on Models and Languages for Coordination of Parallelism and Distribution*, LNCS, Berlin, 1994. Springer Verlag.
- [44] F. Mondada, E. Franzi, and P. Ienne. Mobile Robot Miniaturization: a Tool For Investigation in Control Algorithms. In *ISER'93*, Kyoto, October 1993.
- [45] J.P. Muller. Formalizing emergent collective behaviours: preliminary report. In *Swiss Workshop on Collaborative Agents*, Lausanne, Switzerland, May 2 1997.
- [46] N. Carriero and D. Gelernter. Linda and Message Passing: What have we learned? Technical report, Yale University, Department of Computer Science, 1993.
- [47] U. Nehmzow. *Experiments in Competence Acquisition for Autonomous Mobile Robots*. PhD thesis, University of Edimburg, 1994.
- [48] H.S. Nwana. Software Agents: an Overview. *Knowledge Engineering Review*, 11(3):205–244, 1996.
- [49] R. Pfeifer. Building Fungus Eaters: Design Principles of Autonomous Agents. In Maes, Mataric, Meyer, Pollack, and Wilson, editors, *Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior*, volume 4 From Animals to Animats, Cambridge, MA, 1996. MIT Press/Bradford Books.
- [50] Z.W. Pylyshyn. *The robot's dilemma, the Frame Problem in Artificial Intelligence*. Ablex, Norwood, NJ, 1988.

- [51] Antony Rawston and Alan Wood. BONITA: A Set of Tuple Space primitives for Distributed Coordination. In R. H. Sprague Jr., editor, *Proceedings of the 30th Hawaii International Conference on System Sciences*, volume 1, Wailea, Hawaii, 1997. IEEE. Minitrack on Coordination Languages, Systems and Applications.
- [52] Ray Seyfarth, Suma Arumugham, and Jerry Bickham. Glenda 1.0, 1994. Available via <ftp://seabass.st.usm.edu/pub/glenda.tar.Z>.
- [53] David B. Skillicorn and Domenico Talia. *Programming Languages for Parallel Processing*. IEEE Computer Society Press, 1995.
- [54] L. Steels. The artificial life route to artificial intelligence. Building situated embodied agents. Hillsdale, NJ, 1994. Lawrence Erlbaum.
- [55] L. Steels. When are robots intelligent autonomous agents? *Robotics and Autonomous systems*, 1995.
- [56] V.S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.
- [57] Robert Tolksdorf. Laura: A Coordination Language for Open Distributed Systems. In *Proc. 13th Distributed Computing Systems*, Pittsburgh, Pennsylvania, 1993. IEEE.
- [58] F.J. Varela, E. Thompson, and E. Rosch. *The embodied mind: Cognitive science and human experience*. MIT Press, Cambridge, MA, 1991.
- [59] S.W. Wilson. Knowledge growth in an artificial animal. In *Proceedings of the first International conference on Genetic Algorithms and Their Applications*, pages 16–23, Hillsdale, NJ, 1985. Lawrence Erlbaum Associates.
- [60] T. Ziemke. Adaptive Behavior in autonomous agents. *To appear in Autonomous Agents, Adaptive Behaviors and Distributed Simulations' journal*, 1997.