STL++: A Coordination Language for Autonomy-based Multi-Agent Systems *

Michael Schumacher, Fabrice Chantemargue, Oliver Krone, Béat Hirsbrunner

University of Fribourg, Computer Science Department, PAI group
Pérolles 3, CH-1700 Fribourg, Switzerland
http://www-iiuf.unifr.ch/pai

Abstract. This paper introduces ECM, a new coordination model, and one of his language binding, STL++. STL++ is aimed at making parallel and distributed programs' construction easier by providing a set of powerful tools that clearly describe the coordination part of an application. These tools, embedded in an object-oriented computation language, enable distributed components to be dynamically reconfigured. STL++ acts also as a coordination language especially well suited for distributed implementations of multi-agent systems. It is aimed at giving basic constructs for the implementation of generic multi-agent platforms. The underlying architecture on which STL++ develops, is based on PT-PVM, a software platform providing rich message passing facilities for lightweight processes (threads), which is itself built on top of PVM.
Keywords: Coordination, Distributed Systems, Threads, Concurrency, Communication Models, Agents, Multi-Agent Systems, PVM, PT-PVM.

1 Introduction

Coordination constitutes a major scientific domain 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. *Autonomous Agents*, a discipline of *Artificial Intelligence* which enjoys a boom since a couple of years, embodies inherent distributed applications. Works coming within *Autonomous Agents* are intended to capitalize on the co-existence of distributed entities, and autonomy-based *Multi-Agent Systems* (MAS) are oriented towards interactions, and collaborative phenomena and autonomy.

Today's state of the art parallel programming models, such as (distributed) shared memory models, data and task parallelism, and parallel object oriented models (for an overview see [61]), are used for implementing general purpose distributed applications. However they suffer from limitations concerning a clear separation of the computational part of a parallel application and the "glue" that coordinates the overall distributed program. Especially these limitations make

* Part of this work is financially supported by the Swiss National Foundation for Scientific Research, grant 20-05026.97 and 21-47262.96

distributed implementations burdensome. To study problems related to coordination, Malone and Crowston [47] introduced a new theory called *Coordination Theory* aimed at defining such a "glue". The research in this area has focused on the definition of several coordination models and corresponding coordination languages, in order to facilitate the management of distributed applications.

Coordination is likely to play a central role in MAS, because such systems are inherently distributed. The importance of coordination can be illustrated through two perspectives. On the one hand, a MAS is built by *objective dependencies* which refers to the configuration of the system and which should be appropriately described in an implementation. On the other hand, agents have *subjective* dependencies between them which requires adapted means to program them, often involving high-level notions such as beliefs, goals or plans.

This paper presents STL++, a new coordination language which is aimed at providing a coordination framework for distributed MAS made up of autonomous agents. It can be considered as a platform which allows to describe the organizational structure or architecture of a MAS, with means to dynamically reconfigure it. It is conceived as a basis for the generic multi-agent platform CODA[2]. STL++ is based on the coordination model ECM which is a model for multi-grain distributed applications.

The rest of this paper is organized as follows. Section 2 briefly reviews coordination theory, models and languages. Section 3 examines the question of coordination in the field of MAS. Section 4 introduces the ECM model and gives a thorough description of STL++ which is an instance of ECM. Section 5 illustrates the application of STL++ to a MAS implementation. Section 6 is dedicated to a discussion of ECM and STL++. In the last section we draw some conclusions and outline future work.

2 Coordination Theory, Models and Languages

Coordination can be defined as the process of *managing dependencies between activities* [47], or, in the field of Programming Languages, as the *process of building programs by gluing together active pieces* [13]. 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 [13] 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 in computer science has to be composed of four components (see also [39]):

1. *Coordination entities* as the processes or agents running in parallel which are subject of coordination;

[2] **Co**ordination for **D**istributed Autonomous **A**gents.

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*.

In [13] the authors state that a coordination language is orthogonal to a computation language and forms 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.

The most prominent representative of this class of new languages is Linda [12] which is based on a *tuple space abstraction* as the underlying coordination model. An application of this model has been realized in Piranha [11] (to mention one of the various applications based on Linda's coordination model) where Linda's tuple space is used for networked based load balancing functionality. The PageSpace [16] effort extends Linda's tuple space onto the World-Wide-Web and BONITA [54] addresses performance issues for the implementation of Linda's `in` and `out` primitives. Other models and languages are based on *control-oriented approaches* (IWIM/Manifold [2] [3], ConCoord [30], Darwin [46], TOOLBUS [7]), *message passing paradigms* (CoLa [29], ACTORS [1]), *object-oriented techniques* (Objective Linda [38], JavaSpace [63]), *multi-set rewriting schemes* (Bauhaus Linda [14], Gamma [4]) or *Linear Logic* (Linear Objects [8]). A good overview on coordination models and languages can be found in [50].

Our work takes inspiration from control-oriented models and tuple-based abstractions, and focuses on coordination for purpose of MAS distributed implementations. In order to stress coordination issues in MAS, we introduce major concepts of MAS and analyze where coordination takes place in such systems. This forms the subject of next section.

3 Coordination in Multi-Agent Systems

3.1 What is an agent?

In this section, we try to sketch the fundamental characteristics of what constitutes an autonomous agent. Avoiding to give a new definition and being conscious of the difficulty of the task because of the great variety of existing agent paradigms, we propose to adopt a definition proposed by Franklin and Graesser [23] which should be completed by the definition of Wooldridge and Jennings [70] (see also an answer to the article of Franklin and Graesser in [69]).

Franklin and Graesser define an autonomous agent as follows: *"An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda*

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

Next to a discussion caused by the presentation of this paper at the time of the conference, Franklin and Graesser proposed a first attempt to give an operational definition of an agent, which should allow to describe an agent in terms of the requirements of the definition given above. The description of an autonomous agent could be made by the description of (1) its environment, (2) its sensing capabilities (i.e. its sensors), (3) its actions (i.e. the changes produced in the environment), (4) its drives (preferences or primative motivators) and (5) its action selection architecture (depending of its internal and external sensing and in the service of drives).

Wooldrige and Jennings give the following definition of an autonomous agent [70]: "*... a hardware or (more usually) software-based computer system that enjoys the following properties:*

- *autonomy: agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state;*
- *social ability: agents interact with other agents (and possibly humans) via some kind of agent-communication language;*
- *reactivity: agents perceive their environment, (which may be the physical world, a user via a graphical user interface, a collection of other agents, the INTERNET, or perhaps all of these combined), and respond in timely fashion to changes that occur in it;*
- *pro-activeness: agents do not simply act in response to their environment, they are able to exhibit goal-directed behaviour by taking the initiative.*"

This definition is considered by Wooldridge and Jennings as a *weak* notion of agency. In contrast to it, they define a *strong* notion of agency, which adds to the weak notion, concepts applied to humans, i.e. mentalistic notions such as knowledge, beliefs, intentions, etc. (see [57] for an example).

The definition of Wooldridge and Jennings completes well the one of Franklin and Graesser, especially by requiring autonomy and communication. Autonomy gives an agent the capacity to make independent decisions. From our point of view, it also means that each agent possesses its proper temporality in the sense that it runs concurrently to other agents (for a discussion about concurrent implementation issues of agent systems, see [15]). Agents should also be able to communicate in some way, be it through the environment or directly to one or more agents. This capacity is essential, because agents are always "living" in an (often structured) environment inhabited by other agents, i.e. in a society of agents which is called a *multi-agent system*. The next section discusses communication in more detail.

3.2 Communication between agents

Communication between agents can be considered as follows:

- The capacity to exchange information with other agents. An agent must have a partner to communicate with, either by having an identifier of the partner or by communicating anonymously through predefined interfaces masking the identity of the receiver.
- The intention or the type of the message; this is often realized using illocutary speech acts [59]. Examples are *request*, *deny*, *propose*, *confirm*.
- A common syntax for expressing the information exchanged.
- A common understanding of a message. This can be achieved if both agents share a common information model (which is often referred to as ontology [26] or ontological commitments); it is the context of the interpretation of symbols.

The research has lead to several agent communication languages (ACL) whose most representative is KQML [20], which is the de facto standard ACL. KQML comes along with KIF (Knowledge Interchange Format [24]) which provides a syntax for message content along with ontologies.

We want to stress that a lot of agent languages, aimed at facilitating the building of agent applications, choose ad hoc solutions for the syntax and semantics of messages. Often, applications do not necessitate to formalize illocutions and ontologies for their systems, but they introduce implicit interpretations of the messages being exchanged.

Exchange of information between agents can be at least reduced to four basic paradigms:

- *Peer-to-peer communication*: messages are sent to well known agents, i.e. agents can identify their partners.
- *Broadcast communication*, i.e. sending a message to everybody.
- *Multicast* communication, i.e. sending a message to a group of agents.
- *Generative communication*: communication is realized through the environment: agents generate persistent objects (messages) in the environment which are sensed (read) by other agents. A message can consist of raw data or can be pre-structured like a tuple.

Most of the time, agent languages realize one of these paradigms. Peer-to-peer is the most used. Generative communication is less known in the agent community. As we will see, STL++ offers peer-to-peer, multicast and generative communication.

We consider also *Anonymous Communication* where agents put and read messages on well defined ports (communication interfaces); these ports are connected to other ports belonging to other agents. This means that an agent has no identifier of other agents. Connections between ports can be done by an external specialized agent (a coordinator agent) or as a result of the matching of ports which depends on ports' characteristics.

3.3 Coordinating agents

Agents have interaction capacities in order to exchange information with other agents within a MAS. Communication is a necessary condition to achieve coordination. Coordination in MAS is a key element for the design of such systems, in order to achieve that the community of agents acts in a coherent manner. Coordination is necessary ([31] and [34]):

- for preventing anarchy or chaos in order to achieve a common global goal (the agents must not be necessarily aware of it);
- for meeting global constraints (e.g. such as to resolve a problem in a specific period of time);
- for exchanging distributed expertise, resources or information;
- for managing dependencies between agents' actions;
- for efficiency.

Due to the necessity of coordination in DAI systems, the research in this field has lead to several techniques. According to Nwana [31], they can be classified in four categories:

1. *Organizational Structuring* techniques [18] [28] [67] [36]: a priori organization patterns are defined in order to implicitly define the agent's responsibilities, capabilities, connectivity and control flow. They are fix long-term relationships between agents. In such systems, master/slave or client/server patterns are typically used.
2. *Contracting* techniques: the Contract Net Protocol [62] [51] [32] [17] is based on a decentralized market structure metaphor.
3. *Multi-agent Planning* techniques: [25] [35] agents build a plan that describes all actions and interactions necessary to achieve their respective goals. Planning can be centralized or decentralized.
4. *Negotiation* techniques (significantest part of DAI research in coordination): "*...negotiation is the communication process of a group of agents in order to reach a mutually accepted agreement on some matter*" [10]. In negotiation techniques, agents reason about beliefs, desires and intentions of other agents [64]. Three categories of techniques have raised (see [31]): (1) game theory-based negotiation [45] [55] [40]; (2) plan-based negotiation [41] [5] [6] [49]; and (3) human-inspired and miscellaneous AI-based negotiation approaches [64] [10] [56] [19].

Most coordination models and languages offer tools to express the coordination part of a distributed application, most of the time restricting themselves to express the communication between the components of an application. DAI has developed some techniques, among which negotiation plays a central role, dealing with high-level concepts coming from the strong notion of agency. With STL++ we intend to profit from the experience acquired in coordination theory in order to offer appropriate means to program distributed agent systems, focusing on the weak notion of agency, at least in a first stage. STL++ can also be viewed as an agent-based software engineering language.

3.4 Languages for Constructing Agent Applications

The research in the agent field has lead to several *agent languages* for facilitating the design and the implementation of agent-based applications, covering weak or strong agency. These languages should fill the gap often encountered between agent theory and agent applications. We give a brief overview of existing agent languages.

Concurrent object-oriented languages are a good basis for implementing MAS, because of the similarity between concurrent objects and agents. Fisher [22] even claims that a MAS is simply a system consisting of concurrently executing components. Agent-oriented Programming and its implementation AgentO [60] are based on a societal view of computation; agents are programmed in terms of their mental state. Thomas has developed PLACA [65] which improves AgentO with planning and communication requests for action via high-level goals. Concurrent MetateM [21] [37] implements concurrently executing agents and communication via asynchronous broadcast and multicast; the language uses temporal logic specification to implement agents. APRIL [48], a process oriented symbolic language, offers multi-tasking, powerful message-passing facilities and pattern-matching, allowing to construct most agent architectures. MAIL [27], which is based on APRIL, implements pre-defined abstractions on which MAS prototypes can be constructed. AgentSpeak [66] allows agent programs to be written and interpreted in a manner similar to that of horn-clause logic programs; it consists of aspects of concurrent-object technology. DAISY [52] is composed of CUBL, a distributed object-oriented language, and MAP, based on CUBL, offering limited communication primitives. HOMAGE [53] is an environment for MAS with object-oriented paradigms. Agents can be distributed on the Internet and communicate through different protocols. MIX [33] is a distributed framework for the cooperation of multiple heterogeneous agents, using a declarative language for describing agents. Telescript [68] is an agent language for constructing mobile agents that execute in virtual locations and can communicate with each other.

4 STL++, a Coordination Language for Multi-Agent Systems

Because of the inherent distributivity of MAS, agent languages should have means to clearly describe the coordination part of a MAS application. STL++ applies theories and techniques known from coordination theory and languages in distributed computing to better formalize communication and coordination in MAS applications. STL++ can be considered as an agent language that focuses on the organizational structure of MAS. It is aimed at giving basic constructs to implement more elaborated agent languages with powerful tools, such as in the planned platform CODA.

STL++ is based on the ECM coordination model presented in the next section. STL++ will then be explained in details.

4.1 Coordination using Encapsulation: ECM

ECM[3] is a model for coordination of multi-grain distributed applications. It uses an encapsulation mechanism as its primary abstraction (blops), offering structured separate name spaces which can be hierarchically organized. Within them active entities can communicate anonymously through connections, established by the matching between the communication interfaces of these entities, thus viewing these last as black boxes.

ECM consists of five building blocks:

1. *Processes*, as a representation of active entities;
2. *Blops*, as an abstraction and modularization mechanism for group of 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 inside a blop;
5. *Connections*, as a representation of connected ports.

Figure 1 gives a first overview of the programming metaphor used in ECM.

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

1. The *Coordination Entities* of ECM are the processes of the distributed application;
2. There are two types of *Coordination Media* in ECM: events, ports, and connections which enable coordination, and blops, the repository in which coordination takes place;
3. The *Coordination Laws* are defined through the semantics of the *Coordination Tools* (the operations defined in the computation language which work on the port abstraction) and the semantics of the interactions with the coordination media by means of events.

An application written using the ECM methodology consists of a hierarchy of blops 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.

Blop. A blop is a mechanism to *encapsulate* a set of objects. Objects residing in a blop are per default only visible within their "home" blop. In Figure 1, two blops are shown. Blops have the same interface as processes, i.e. a name and a possibly empty set of ports, and can be hierarchically structured. Blops are an abstraction for an agglomeration of objects to be coordinated and serve as a separate name space for port objects, processes, and subordinated blops as well as an encapsulation mechanism for events.

[3] **E**ncapsulation **C**oordination **M**odel.

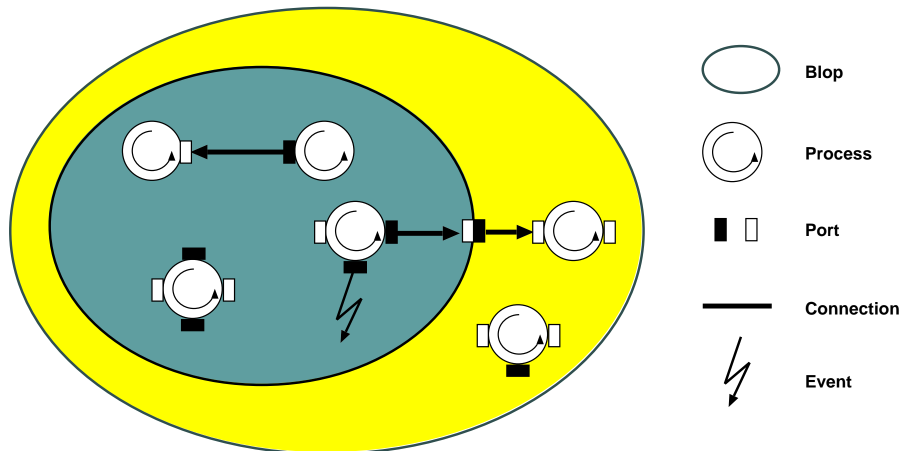

Fig. 1. The Coordination Model of ECM.

Processes. A process in ECM is a typed object, it has a name and a possibly empty set of ports. Processes in the ECM model do not know any kind of process identification, instead a black box process model is used. A process does not have to care about to which process information will be transmitted or received from. Process creation and termination is not part of the ECM model and is to be specified in the instance of the model.

Ports. Ports are the interface of processes and blops to establish connections to other processes/blops, i.e. communication in ECM is handled via a connection and therefore over ports. Ports have *names* and a set of well defined *features* describing the port's characteristics. Names and features of a port are referred to as the port's *signature*. The combination of port features results in a port type.

Port Features. Features of pairs of ports must comply with each other for ports to match. Ports are characterized through the communication paradigm they support, namely point-to-point stream communication (with classical message-passing semantics), closed group (with broadcast semantics) and blackboard communication. The communication feature is mandatory and must be supported by all ECM realizations, that is, each language binding of the ECM model must provide means to specify a minimal set of port features.

Port Matching. The matching of ports is defined as a relation between port signatures. Four general conditions must be fulfilled for two ports to get matched: (1) both use the same communication paradigm; (2) both have the same name; (3) both belong to the same level of abstraction, i.e., are visible within the same hierarchy of blops; and (4) both belong to different objects (process or blop).

Conceptually the matching of process ports can be described as follows. When a process is created in a blop, it creates with its port signature a "potential"

in the blop where it is currently embedded. If two compatible potentials exist in the blop, and if conditions (1)-(4) are fulfilled, the connection between the corresponding ports is established and the potentials disappear.

Connections. The matching of ports results in the following connections:

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

Events. Events can be attached to conditions on ports of blops or processes. These conditions will determine when the event will be triggered in the blop. Condition checking is implementation dependent (see STL++'s event definition as an example of how to define event semantics on ports).

4.2 The Coordination Language STL++

We designed and implemented a first language binding of the ECM model, called STL[4]. STL is a realization of the ECM model applied to multi-threaded applications on a LAN of UNIX workstations. STL materializes the separation of concern as it uses a separate language exclusively reserved for coordination purposes and provides primitives which are used in a computation language to interact with the entities. The implementation of STL is based on PT-PVM [43], a library providing message passing and process management facilities at thread and process level for a cluster of workstations. In particular blops are represented by heavy-weight UNIX processes, and ECM processes are implemented as light-weight processes (threads). More details about STL can be found in [42].

However, it turned out that the separation of *code* can not always be maintained. Although the black box process model of ECM is a good attempt to separate coordination and computation code, dynamic properties proved to be difficult to express in a separate language. This is for example reflected in STL by the primitives which must be used in the computation language in order to use dynamic coordination facilities of STL. Dynamic properties can not be separated totally from the actual program code. Furthermore, a duplication of code for processes may introduce difficulties to manage code for a distributed application. These observations lead us to the development of a new coordination language, called STL++.

Starting from the experience acquired with STL, STL++ implements the conceptual model of ECM by enriching a given object oriented language (C++)

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

with coordination primitives. It allows us to take advantage of some of the well known object oriented programming concepts, namely inheritance, polymorphism, modularity, strong typing and templates. The single language approach of STL++ separates *conceptually* the coordination from the computational part of an application. This contrasts with STL which uses two languages, and hence supports a separation at code level.

A STL++ application is a set of classes that inherit from the base classes of the library (see Figure 2). The main class of an application must inherit from `World`, which is the default blop containing all other entities.

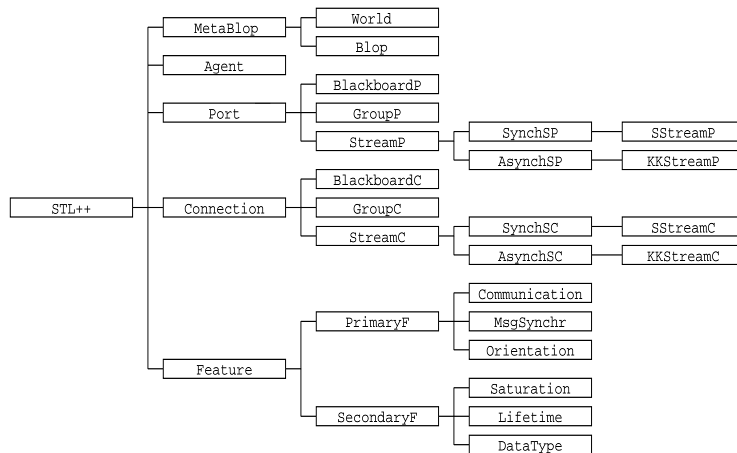

Fig. 2. Simplified class hierarchy in STL++

Blops. A blop must be declared as a class, which inherits from the base class `Blop`. The user must call `initBlop()` in order to initialize the blop. The creation of a blop results in the initialization of all enclosing blops, ports and agents (as ECM processes are named in STL++). Blops can be parameterized like normal objects in C++.

As manager of its enclosed entities, the blop creates agents as well as events, and binds events to ports. In STL++, new blops can be dynamically created during execution of a program. Blops are implemented as heavy-weight UNIX processes (like in STL).

Agents. In STL++, ECM processes are named agents. They are objects, which are instances of the base class `Agent`. They can be created directly by blops, through events or by other agents. An agent object must be initialized by the user with `initAgent()`.

For communication purposes, agents can dynamically create ports of predefined types through instantiation of a port class.

Agent termination is implicit: the agent disappears along with its ports and data, when the corresponding object terminates. Regarding the implementation, an `Agent` object is embedded in a light-weight process (thread).

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

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

1. `Communication`. This feature captures ECM communication paradigms;
2. `Msg Synchronization`. This feature gives to the connection the usual semantics of message passing communication. Possible values are `synchron` and `asynchron`;
3. `Orientation`. This feature defines the direction of the data flow over a connection: there are three possible values, namely `in` (in-flowing), `out` (outflowing) and `inout` (bi-directional).

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

1. `Saturation`. This feature, ranging from 1 to `INF` (infinity) by integer values, defines the number of connections a port can have;
2. `Lifetime`. This feature, ranging from 1 to `INF` (infinity) by integer values, indicates the number of data units that can pass through a port before it decays;
3. `Data Type`. This feature defines the type of data authorized to pass through a port. Possible types can be basic data types, tuples of basic data types or undefined.

Filters on ports can be defined. They can range from boolean functions testing specific properties on values flowing through ports, to transformation routines modifying values.

Matched ports result in connections which are first-class entities. In STL++, four basic port types are defined, corresponding to the combinations of the primary features as displayed in Table 1. Note that, provided that ports match, it yields four basic connection types.

`Blackboard` **port type** The resulting connection, as a result of the matching of ports of this type, has a blackboard semantics. The number of participating ports is unlimited. Messages are persistent objects which can be retrieved using a symbolic name. An agent reading/writing a message does not have to be aware of the agent that has written or that will read this message. Moreover, messages are not ordered. To access the blackboard, the following Linda-like primitives are provided: `put()` (non-blocking), `get()` (blocking) and `read()` (blocking). Blackboard connections are persistent: if all the ports involved in a blackboard

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

Table 1. Basic port types with their primary feature's values.

connection disappear, the connection still persists in the blop space with all the information it carries, so that new ports can later on reconnect to the blackboard and recover the pending information.

`Group` **port type**. The resulting connection has a closed group semantics. The number of participating ports is unlimited. Each member of the group can broadcast asynchronously messages to every participant in the group. Messages are stored at the receiver side. Thus, if a port in a group disappears, then the sequence of information that has not been read is lost. The primitives for accessing the group are: `get()` (blocking) and `put()` (non-blocking).

`S-Stream` **port type**. The semantics of a connection resulting from the matching of two S-Stream ports (S for synchronous) has the same semantics as the S-Channel defined in [2], in particular this connection is uni-directional. This connection always results from the matching of contradictory oriented ports, namely a producer and a consumer (see Table 1). In contrast to other connections, this connection never contains data, due to its synchronous nature. So the destruction of the producer or the consumer never causes loss of data. The primitives for accessing the port are `get()` (blocking) and `put()` (blocking).

`KK-Stream` **port type**. The semantics of a connection resulting from the matching of two KK-Stream ports (K for keep) is analogous to the asynchronous KK-Channel defined in [2], with its specific semantics when a port disappears from one end of the connection. As for S-Streams, this connection always results from the matching of contradictory oriented ports. This connection is not disconnected from one side if it is disconnected from the other side, therefore it is persistent. Another port can later on reconnect to the connection. If the connection is broken at its consuming port, the next new matching port will consume all pending data. If the connection is broken at its producing port, the consuming port will be able to continue to consume all data in the connection. The primitives for accessing the port are `get()` (blocking) and `put()` (non-blocking).

Communication between Agents. Communication between agents is realized through connections which are the result of matched ports. In accordance with the ECM model, the matching is realized as a relation between port signatures. In STL++, ports to match, must belong to the same blop and must comply at name and at feature levels: (1) *Name level.* A port may have several names; in this case, each name belongs to a different connection. Two ports match at name level if they share at least one name; (2) *Feature level.* Two de-

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

Table 2. Compatibility for Features F for two Ports P1 and P2

grees of matching are considered: *exact* and *plug-in* matching. They are described hereafter.

We define for each feature `f` a member function `compatible(Feature *f)` that verifies if another feature is *compatible* to this one.

For new secondary features, the user has to implement an appropriate member function `compatible(Feature *f)` which returns `true` if `f` is compatible to this feature. This mechanism allows to introduce new secondary features very easily. Table 2 gives an overview of the already present compatibility functions used by the STL++ runtime system.

If for two ports all secondary features are identical in order to match, we call this *exact matching*; any other relationship between port features is called *plug-in matching*.

By introducing several names for each port, STL++ allows a port to be connected to different connections. Figure 3 shows three examples: a) an output port is connected to two other input ports; b) a group port is connected to two group connections; c) a blackboard port is connected to two blackboard connections. Data written on such multiple connection ports are echoed on every connection. For stream connections, `1:1`, `1:n`, `n:1` and `n:m` communication patterns can be built. Likewise, several blackboards can be connected to a single port. The same can be done with group ports.

Events. An event is a member function of a blop which will be triggered by a condition function attached to a port object.

Conditions on ports are verified either when data flow through the port or when a blop/agent accesses it. Agents can also directly raise events defined in their blop.

Events are instantiated with a specific lifetime which determines how many times they can be triggered. After an event with lifetime 1 has been triggered, a blop is not tuned anymore to handle subsequent events of the same type; in order to handle such an event again, the event handling routine must be re-installed which is usually done in the event handling routine of the event currently processed.

The following predefined conditions trigger an event:

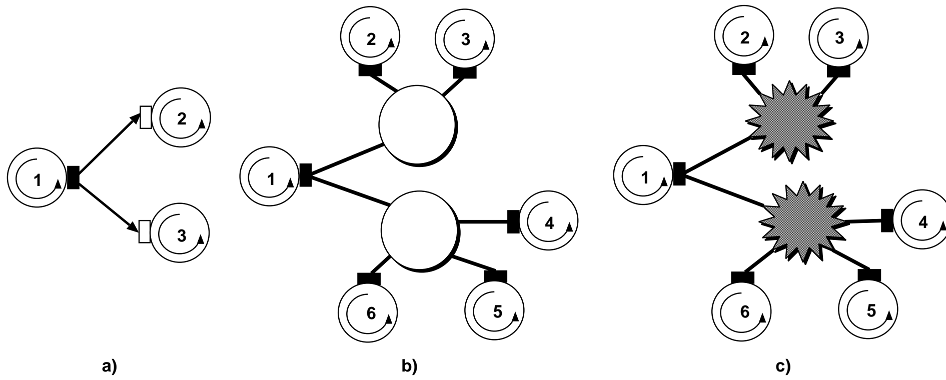

Fig. 3. Ports with multiple connections: a) stream b) group c) blackboard

– `accessed(p)`. The port has been accessed;
– `unbound(p)`. The port does not belong to any connection;
– `saturated(p)`. The port is saturated;
– `msg-handled(p, int n)`. `n` messages have been handled;
– `terminated(p)`. The port has lasted longer than its lifetime.

5 Application

5.1 The Framework

In order to illustrate how STL++ can be used, we present a specific model of MAS which adresses a range of applications in the framework of collective robotics and their simulations.

The model 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* contains a list of *Neighbour Cells*, which implicitly sets the topology, and the set of objects actually available on it at a given time.

The architecture of an agent is displayed on figure 4. An agent possesses some sensors to perceive the world within which it moves, and some effectors to act in this world, so that it conforms with the definition of agents presented in section 3.1. Moreover, in the context of New Artificial Intelligence [9], it complies with the prescriptions of *physically embodied agents* and *simulated embodied agents* [71]. The implementation of the different modules presented on Figure 4, 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 . The *State* module encompasses the private information of the agent . The *Actions* module typically consists of the basic actions the agent can take . 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 [71]. Operational autonomy is defined as the capacity to operate without human intervention, without being remotely controlled. 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.

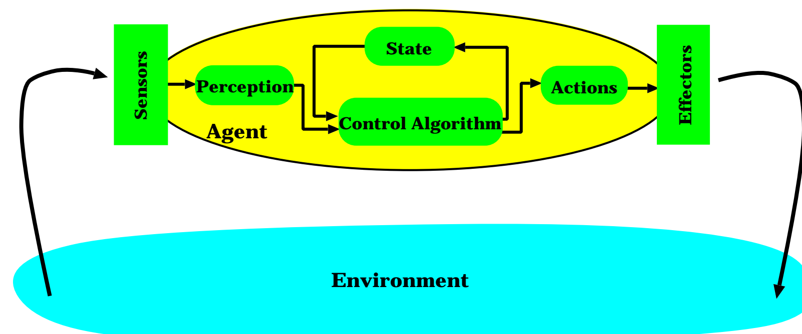

Fig. 4. Architecture of an agent.

The specific application we consider tackles a quite common problem in collective robotics which is still given a lot of consideration: agents are in charge of regrouping objects distributed in their environment. The innovative aspect of our approach rests indeed on a system integrating operationally autonomous agents: every agent in the system has the freedom to act on a cell (the agent decides by itself which action to take). Therefore, there is not in the system any type of master responsible for supervising the agents, nor any type of cooperation protocol, thus allowing the system to be more flexible and fault tolerant.

5.2 A Preliminary STL++-based Implementation

For this preliminary implementation, the *Environment* is made up of a torus grid with a four connectivity (each cell has four neighbors). Agents comply rigorously with the model previously introduced (Figure 4). They sense the environment through their sensors and act upon their perception at once.

To put to good use distributed systems, the *Environment* is split into sub-environments, each of which being handled by a blop, as indicated on Figure 5, 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.

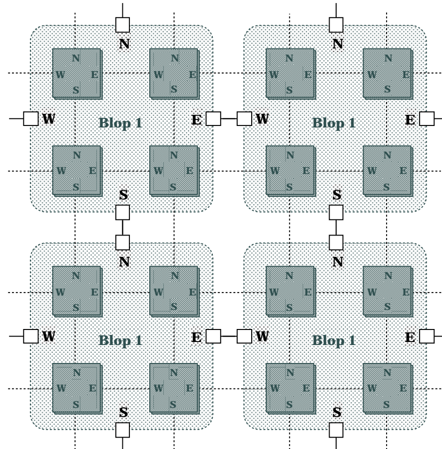

Fig. 5. Splitting an environment made up of cells into four blops.

We introduce several variants of the `KKStreamP` port class: `KKStreamP_i` and `KKStreamP_o` ports are respectively input and output `KKStreamP` ports with `saturation` set to `N`; `KKStreamP_iN` and `KKStreamP_oN` are respectively input and output `KKStreamP` ports with `saturation` set to `N`.

The implementation distinguishes two types of agents: (1) agents used for purpose of implementation on a distributed platform (*init*, *initSimRobot* and *taxi*) and (2) agents intrinsic to the MAS application (*simRobot* and *subEnv*).

Global Structure. The meta-blop *world* is composed of an *init* agent, responsible for the global initialization of the system, and a set of predefined blops (called *se*), each one handling a sub-environment.

Init has two ports (of type `KKStreamP_o`) for every blop to be initialized (Figure 6). The rôle of *init* is twofold: first, to create through its *cre_SimRbs* port the initial agents *simRobots* within every blop; secondly, to set up through its *cre_SubEnv* port the sub-environment (size, number of objects, etc.) of every blop.

Blop *se*. Figure 6 shows the basic organization of STL++ agents within a blop *se* and their coordination through ports. The following types of agents are present: *initSimRobot*, *taxi*, *subEnv* and *simRobot* agents.

Each blop has ten static ports: four `KKStreamP_o` *out-flowing direction* ports (*north_o, south_o, west_o, east_o*) and four `KKStreamP_i` *in-flowing direction* ports (*north_i, south_i, west_i, east_i*), which are used for *simRobot* migration, and two `KKStreamP_i` ports, namely *i_SimRobot* and *i_SubEnv* used respectively for the creation of the initial *simRobots* and for the initialization of the *subEnv* agent.

For the time being, the topology between blops is set in a static manner, by creating the ports with appropriate names. The four *in-flowing direction* ports of a blop match with ports of its inner agent *initSimRobot*. The four *out-flowing*

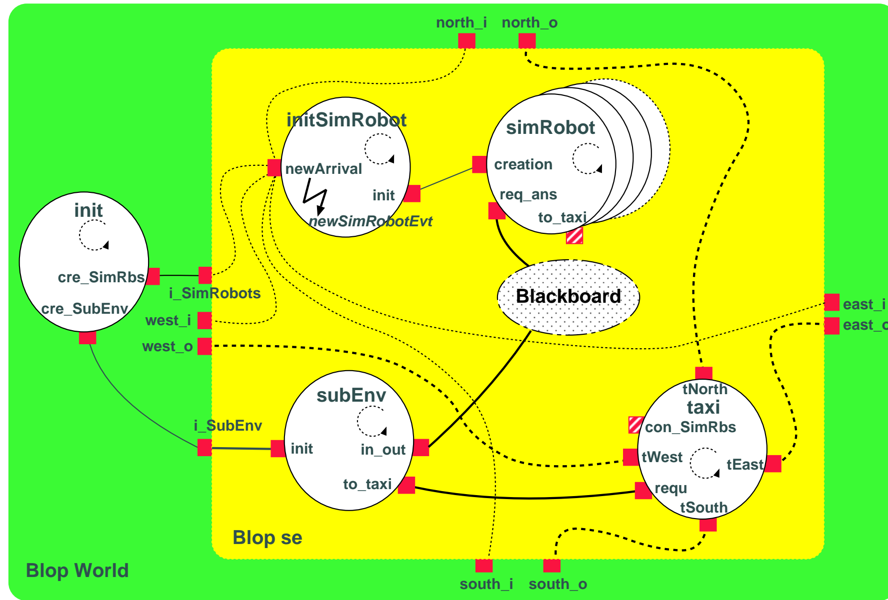

Fig. 6. *init* agent and blop *se*: solid and dotted lines are introduced just for a purpose of visualization

direction ports of a blop match with ports of its inner agent *taxi*.

initSimRobot **Agent,** *newSimRobotEvt* **Event**. The *initSimRobot* agent is responsible for the creation of *simRobots*. It has two static ports: *newArrival* and *init*. The *newArrival* `KKStreamP_iN` port is connected to all *in-flowing direction* ports of the blop within which it resides. As soon as a value comes to this port, *initSimRobot* copies it onto its *init* `KKStreamP_oN` port. In the meantime, the *newSimRobotEvt* event is triggered and it will create a new *simRobot* agent, which through its *creation* port will read the value that was previously written on the *init* port of the *initSimRobot* agent. Transmitted values are for instance the *state* of the agent to create.

simRobot **Agent**. This agent (code in Figure 7) has two static ports (*req_ans* of type `BlackboardP` and *creation* of type `KKStreamP_i`) plus *to_taxi* a dynamic `KKStreamP_o` port. As already stated, this agent reads on its *creation* port some values (its *state*). All *req_ans* ports of the *simRobot* agents are connected to a *Blackboard*, through which these agents will sense their environment (*perception*) and act into it (*action*), by performing *put*/*get* operations (Linda-like *out*/*in*) with appropriate messages. The type of action depends on the type of *control Algorithm* implemented within the agent (see Figure 4). The *to_taxi* port is used to communicate dynamically with the *taxi* agent in case of migration: the *state* of the agent *simRobot* is indeed copied to the *taxi* agent. The decision of migrating is always taken by the *subEnv* agent.

subEnv **Agent**. The *subEnv* agent handles the access to the sub-environment

```
class simRobot:Agent {
public:
  simRobot();
  ~simRobot(){};
private:
  BlackboardP *req_ans;
  KKStreamP_i *creation;
  KKStreamP_o *to_taxi;
}


simRobot::simRobot() {
  ByteTempl<32> state, answer;
  ByteObject<32> *req;
  initAgent();
  req_ans = new BlackboardP("SUBENV-SIMROBOT");
  creation = new KKStreamP_i("SIMROBOT-INIT");
  Msg stateTp(state);                          // Message
  boolean noMigration = TRUE;
  creation->get(0, stateTp);                   // Initialize

  while (noMigration) {                        // Perception/Action
    req = make_req();
    Msg requestTp("request", req->id, *req);
    req_ans->put(0, requestTp);                // Put request
    Msg answerTp("answer", req->id, answer);
    req_ans->get(0, answerTp);                 // Get answer
    control(answer);                           // Control Algorithm
    state = update_state(answer);
    noMigration = migrate_p(answer);
  }
  to_taxi = new KKStreamP_o("MIG" + req->id); // For migration
  to_taxi->put(0, stateTp);                    // Transfer state
  exit(0);                                     // To taxi
}
```

Fig. 7. Implementation of class *simRobot*.

and is in charge of keeping data consistency. It is also responsible for migrating *simRobot* agents, which will cross the border of a sub-environment. It has a static *in_out* port (of type `BlackboardP`) connected to the *Blackboard* and a static `KKStreamP_o` port *to_taxi* connected to the *taxi* agent. Once initialized through its *init* `KKStreamP_i` port, the *subEnv* agent builds the sub-environment. By performing *put*/*get* operations with appropriate tuples, *subEnv* will process the requests of the *simRobot* agents (e.g. number of objects on a given cell, move to next cell) and reply to their requests (e.g. x objects on a given cell, move registered). When the move of a *simRobot* agent will lead to cross the border (cell located in another blop), *subEnv* will first inform this agent it has to migrate and then inform the *taxi* agent a *simRobot* agent has to be migrated (the direction the agent has to take will be transmitted).

The *taxi* Agent. The *taxi* agent is responsible for migrating *simRobot* agents across blops' boundaries. It has four static *direction* ports (of type `KKStreamP_o`), which are connected to the four *out-flowing direction* ports of the blop within

which it stands. When *taxi* receives on its static `KKStreamP_i` port *requ* the direction towards where the *simRobot* agent has to migrate, it will create a dynamic `KKStreamP_i` port *con_SimRbs* in order to establish with the appropriate *simRobot* agent 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 the *newArrival* port of the *initSimRobot* agent of the concerned blop inducing the dynamic creation of a new *simRobot* agent in the blop, thus materializing the migration.

6 Discussion

6.1 STL++ as a coordination language

As a coordination language for distributed programming, ECM along with STL++ present some similarities with several coordination languages, and particularly with the IWIM model [2] and its instantiation MANIFOLD [3]. However they differ in several important points:

- One might be inclined to identify blops with IWIM managers (manifolds). This is not the case, because blops are not coordinators that create explicitly interconnections between ports. The establishment of connections is implicit, resulting from a matching mechanism, depending on the types and the states of the ports. This is definitely a different point of view in which communication patterns are not imposed. Furthermore, the main characteristics of blops is to encapsulate objects, thus forming a separate name-space for enclosed entities and an encapsulation mechanism for events. Nested blops are a powerful mechanism to structure private name-spaces, offering a hierarchical model which is more explicit than in IWIM.
- ECM generalizes connection types: either stream, blackboard or group. This adds powerful means to express coordination with tuple-space models and does not restrict to channels. Refined semantics can be defined in virtue of port characteristics (features).
- In ECM, events are not signals broadcast in the environment, but routines belonging to blops. They are attached to ports with conditions on their state that determine when events are launched. Events can create new blops, processes (agents) or ports, and attach events to ports. Their action area is limited to a blop.
- Interconnections evolve through configuration changes of the set of ports within a blop, induced by events and also by the processes (agents) themselves. In fact, the latter can create other processes (agents), and create and destruct ports, thus yielding to communication topology changes.
- Instantiations of the ECM model implement a separate coordination language for STL (as for Manifold) or integrate the model in an existing computation language for STL++.
- Concerning the range of applications, STL++ is especially adapted for distributed artificial intelligence.

ECM and STL++ present similarities with several other coordination models and languages like Linda [12], Darwin [46] or ConCoord [30]. We mention few other specific characteristics of our model. Like several further developments of the Linda model (for instance Objective Linda [38]), ECM uses a hierarchical multiple coordination space model, in contrast to the single flat tuple space of the original Linda. Processes (agents) get started through an event in a blop, or automatically upon initialization of a blop, or through a creation operation by another process (agent); Linda uses one single mechanism: `eval()`. Processes (agents) do not execute in a medium which is used to transfer data. In order to communicate, they do not have references to other processes (agents) or to ports belonging to other entities; they communicate anonymously through their ports.

6.2 Implementing coordination in MAS with STL++

Several DAI coordination techniques such as multi-agent planning and negotiation are designed for strong agency applications, comprising the exchange of high-level information such as plans, knowledge, beliefs or intentions. Thus they do not treat basic coordination involving the communication topology between agents, which constitutes the basis of more complex strategies. Organizational structuring techniques try to give this basis by supplying an a priori organization by long-term relationships between agents. This technique has shown good results, especially with master/slave or client/server patterns, sometimes using blackboard architecture. STL++ tries to resolve weaknesses encountered in organizational structuring, especially by offering means for dynamical reconfiguration, and by using semi-locality for the management of agents (in blops), thus avoiding centralization. STL++ also offers a complete set of communication primitives, ranging from peer-to-peer, group to blackboard communication.

STL++ can also be considered as an agent language for constructing agent applications, stressing on the coordination part of a MAS. It is comparable to several other agent languages. It must be stressed that STL++ is directly embedded in an object-oriented language and not realized with high-level syntaxical constructs, as it is done for many other agent languages. STL++ is close to APRIL [48] in the sense that it has similar goals, namely to be conceived as a platform oriented to the implementation of enhanced multi-agent systems. In STL++ the management of messages is simple: they are tuples. Filters on ports are used to achieve the functionality of pattern matching on messages. Concurrent MetateM [21] shares also some features with STL++: the latter enhances communication with blackboard and peer-to-peer (synchron and asynchron) paradigms.

STL++ has some characteristics that make it especially aimed at implementing MAS. Most important are the following:

- Blops constitute a mechanism which allows to construct a structured set of different environments, each of which is a closed private coordination space with communication interfaces to other environments. A set of complex interconnected and hierarchically organized spaces is often desirable in MAS.

- Realizing agents as black boxes is a good way to implement autonomous agents. An agent owns exclusive control over its internal state and behaviour; it can define by itself its ports. It is seen from external views (its environment) as a delineated entity presenting clear interfaces. Being distinct from the outside (the environment), an agent is embodied in its environment composed by its surrounding blop and the agents living in it.
- Ports are a adequate way to implement sensing and acting capacities of agents. In fact, agents should perceive and act through different means, which is enabled in virtue of the possibility to define port types and several instantiations of them. A port perceives specific data; this is enforced with the data type feature of ports and the filter mechanism. Furthermore, filtering provides some easy means to limit perception to a range of values, or to retrieve only selected information.
- As agents and environments are aimed at evolving in time, dynamicity is an important point in MAS. This leads the design of STL++ to allow reorganizing blops and to create and destruct new ports, thus yielding to new communication topologies (configuration); an agent is able to adapt its sensing.
- With autonomy comes the notion of proper temporality. In its implementation, STL++ try to fullfil this requested point by embedding agents in light-weight processes.

7 Conclusion

In résumé, we presented ECM a coordination model and an instantiation, namely STL++, a platform to implement MAS by expressing the organizational structure and the communication between agents.

A first prototype of STL++ is being developed in C++ on top of PT-PVM [43], a software platform providing rich message passing facilities for light-weight processes (threads), which is itself built on top of PVM.

STL++ is still to be extended in order to encompass as many generic coordination patterns as possible, yielding in templates at disposal for general purpose implementations. Future works will consist in: (1) carrying on the implementation of the STL++ library and (2) developing a graphical user interface to facilitate the specification of the coordination part of a distributed application.

The mapping of ECM blops onto heavy-weight UNIX processes and ECM processes (agents) onto threads is not the only possible realization of the ECM model of coordination. We are currently working on an instantiation of the ECM model to coordinate applications on the World Wide Web [58] [44], where ECM blops are used to represent a LAN of a WAN application and ECM processes the physical machines of a LAN.

References

1. G. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

2. F. Arbab. The IWIM Model for Coordination of Concurrent Activities. In Paolo Ciancarini and Chris Hankin, editors, *Proceedings of the First International Conference on Coordination Models, Languages and Applications*, number 1061 in LNCS. Springer Verlag, April 1996.
3. F. Arbab, I. Herman, and P. Spilling. An Overview of Manifold and its Implementation. *Concurrency: Practice and Experience*, 5(1):23–70, February 1993.
4. J.P. Banâtre and D. Le Métayer. Programming by Multiset Transformation. *Communications of the ACM*, 36(1):98–111, 1993.
5. M. Barbuceanu and M.S. Fox. The Design of a Coordination Language for Multi-Agent Systems. In J.P. Muller, M.J. Wooldridge, and N.R. Jennings, editors, *Intelligent Agents III. Proceedings of Third International Workshop on Agent Theories, Architectures, and Languages (ATAL'96))*, number 1193 in LNAI. Springer Verlag, August 1996.
6. M. Barbuceanu and M.S. Fox. Integrating Communicative Action, Conversations and Decision Theory to Coordinate Agents. In *Agent'97 Conference Proceedings*, Marina del Rey, California, February 1997.
7. J.A. Bergstra and P. Klint. The ToolBus Coordination Architecture. In Paolo Ciancarini and Chris Hankin, editors, *Proceedings of the First International Conference on Coordination Models, Languages and Applications*, number 1061 in LNCS. Springer Verlag, April 1996.
8. 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.
9. R.A. Brooks. Intelligence without Representation. *Artificial Intelligence, Special Volume: Foundations of Artificial Intelligence*, 47(1-3), 1991.
10. S. Bussmann and J. Muller. A Negotiation Framework for Co-operating Agents. In S. M. Deen, editor, *Proceedings of CKBS-SIG*, pages 1–17. Dake Centre, University of Keele, 1992.
11. N. Carriero, E. Freeman, D. Gelernter, and D. Kaminsky. Adaptive Parallelism and Piranha. *IEEE Computer*, 28(1), January 1995.
12. N. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, 1989.
13. N. Carriero and D. Gelernter. Coordination Languages and Their Significance. *Communications of the ACM*, 35(2):97–107, February 1992.
14. 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.
15. F. Chantemargue, O. Krone, M. Schumacher, T. Dagaeff, and B. Hirsbrunner. Autonomous Agents: from Concepts to Implementation. In *Proceedings of the Fourteenth European Meeting on Cybernetics and Systems Research (EMCSR'98)*, volume 2, pages 731–736, Vienna, Austria, April 14-17 1998.
16. P. Ciancarini, A. Knoche, R. Tolksdorf, and Fabio Vitali. PageSpace: An Architecture to Coordinate Distributed Applications on the Web. In *Proceedings of the Fifth International World Wide Web Conference*, volume 28 of *Computer Networks and ISDN Systems*, 1996.
17. S.E. Conry, R.A. Meyer, and V.R. Lesser. Multistage Negotiation in Distributed Planning. Technical report, COINS, University of Massachusetts, Amherst Boston, 1986.

18. E.H. Durfee, V.R. Lesser, and D.D. Corkill. Coherent Cooperation among Communicating Problem Solvers. *IEEE Trans Comput*, 36(11):1275–1291, 1987.
19. E.H. Durfee and T.A. Montgomery. A Hierarchical Protocol for Coordinating Multi-agent Behaviour. In *Proceedings of the 8th National Conference on Artificial Intelligence*, pages 86–93, Boston, MA, 1990.
20. T. Finin, R. Fritzon, D. McKay, and R. McEntire. KQML as an Agent Communication Language. In *Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM)*. ACM Press, 1994.
21. M. Fisher. A Survey of Concurrent MetateM - the Language and its Applications. In D.M. Gabbay and H.J. Ohlbach, editors, *Temporal Logic - Proceedings of the First International Conference*, number 827 in LNAI, pages 480–505. Springer-Verlag, 1994.
22. M. Fisher. Representing and Executing Agent-Based Systems. In M. Wooldridge and N.R. Jennings, editors, *Intelligent Agents. Proceedings of First International Workshop on Agent Theories, Architectures, and Languages (ATAL'94)*, number 890 in LNAI. Springer Verlag, August 1994.
23. S. Franklin and A. Graesser. Is it an Agent or just a Program? A Taxonomy for Autonomous Agents. In J.P. Muller, M.J. Wooldridge, and N.R. Jennings, editors, *Proceedings of ECAI'96 Workshop (ATAL). Intelligent Agents III. Agent Theories, Architectures, and Languages*, number 1193 in Lectures Notes in Artificial Intelligence, pages 21–35, August 1996.
24. M.R. Genesereth and R.E. Fikes. Knowledge Interchange Format, Version 3.0. Reference Manual. Technical Report Logic-92-1, Computer Science Department, Stanford university, 1992.
25. M. Georgeff. Communication and Interaction in Multi-agent Planning. In *Proceedings of the 1983 National Conference on Artificial Intelligence*, pages 125–129, 1983.
26. T.R. Gruber. A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, 5:119–220, 1993.
27. H. Haugeneder, D. Steiner, and F.G. McCabe. IMAGINE: A Framework for building Multi-agent Systems. In S. M. Deen, editor, *Proceeding of the 1994 International Working Conference on Cooperating Knowledge Based Systems (CKBS-94)*, pages 31–64, 1994.
28. B. Hayer-Roth. A Blackboard Architecture for Control. *Artificial Intelligence*, 25:251–321, 1985.
29. B. Hirsbrunner, M. Aguilar, and O. Krone. CoLa: A Coordination Language for Massive Parallelism. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, Los Angeles, California, August 14–17 1994.
30. A.A. Holzbacher. A Software Environment for Concurrent Coordinated Programming. In Paolo Ciancarini and Chris Hankin, editors, *Proceedings of the First International Conference on Coordination Models, Languages and Applications*, number 1061 in LNCS. Springer Verlag, April 1996.
31. L. Lee H.S. Nwana and N.R. Jennings. Co-ordination in Multi-Agent Systems. In H. S. Nwana and N. Azarmi, editors, *Software Agents and Soft Computing*, number 1198 in LNAI. Springer Verlag, 1997.
32. M. Huhns and M.P. Singh. Ckbs-94 tutorial. In *Distributed Artificial Intelligence for Information Systems.* Dake Centre, University of Keele, 1994.
33. C.A. Iglesias, J.C. Gonzalez, and J.R. Velasco. MIX: A General Purpose Multiagent Architecture. In M. Wooldridge, J. P. Mueller, and M. Tambe, editors, *Intelligent Agents II. Proceedings of Second International Workshop on Agent Theories,*

Architectures, and Languages (ATAL'95), number 1037 in LNAI. Springer Verlag, August 1995.
34. N.R. Jennings. Coordination Techniques for Distributed Artificial Intelligence. In G.M.P. O'Hare and N.R. Jennings, editors, *Foundations of Distributed Artificial*. John Wiley and Sons, 1996.
35. Y. Jin and T. Koyoma. Multi-agent Planning through Expectation-based Negotiation. In *Proceedings of the 10th Int Workshop on DAI*, Texas, 1990.
36. P. Kearney, A. Sehmi, and R. Smith. Emergent Behaviour in a Multi-agent Economics Simulation. In A. G. Cohn, editor, *Proceedings of the 11th European Conference on Artificial Intelligence*. John Wiley, 1994.
37. A. Kellet and M. Fisher. Concurrent MetateM as a Coordination Language. In D. Garlan and D. Le Metayer, editors, *Proceedings of the Second International Conference on Coordination Models, Languages and Applications (Coordination'97)*, number 1282 in LNCS. Springer Verlag, September 1997.
38. T. Kielmann. *Objective Linda: A Coordination Model for Object-Oriented Parallel Programming*. PhD thesis, Dept. of Electrical Engineering and Computer Science, University of Siegen, Germany, 1997.
39. T. Kielmann and G. Wirtz. Coordination Requirements for Open Distributed Systems. In *Proceedings of PARCO'95*. Elsevier, 1996.
40. S. Kraus and J. Wildenfield. The Function of Time in Cooperative Negotiations: Preliminary Report. Technical report, Department of Computer Science, University of Maryland, 1991.
41. T. Kreifelt and F. von Martial. A negotiation Framework for Autonomous Agents. In Y. Demazeau and J. P. Mueller, editors, *Proceedings of Decentralized AI2*. Elsevier Science, 1991.
42. O. Krone. *STL and Pt-PVM: Concepts and Tools for Coordination of Multithreaded Applications*. PhD thesis, University of Fribourg, 1997.
43. O. Krone, B. Hirsbrunner, and V.S. Sunderam. PT-PVM+: A Portable Platform for Multithreaded Coordination Languages. *Calculateurs Parallèles*, 8(2):167–182, 1996.
44. S.B. Lamine, J. Plaice, and P. Kropf. Problems of computing on the WEB. In A. Tentner, editor, *High Performance Computing*, pages 296–301, Atlanta, Georgia, USA, 1997.
45. R.D. Luce and H. Raiffa. *Games and Decisions*. John Wiley and Sons, 1957.
46. J. Magee, N.Dulay, and J. Kramer. Structuring parallel and distributed programs. *Software Engineering Journal*, pages 73–82, March 1993.
47. T.W. Malone and K. Crowston. The Interdisciplinary Study of Coordination. *ACM Computing Surveys*, 26(1):87–119, March 1994.
48. F.G. McCabe and K.L. Clark. April - agent process interaction language. In M. Wooldridge and N.R. Jennings, editors, *Intelligent Agents. Proceedings of First International Workshop on Agent Theories, Architectures, and Languages (ATAL'94)*, number 890 in LNAI. Springer Verlag, August 1994.
49. A.I. Mouaddib. Progressive Negotiation For Time-Constrained Autonomous Agents. In *Agent'97 Conference Proceedings*, Marina del Rey, California, February 1997.
50. G.A. Papadopoulos and F. Arbab. Coordination Models and Languages. In M. Zelkowitz, editor, *Advances in Computers, The Engineering of Large Systems*, volume 46. Academic Press, August 1998.
51. H.V.D. Parunak. Manufacturing Experiences with the Contrat Net. In L. Gasser and M. Huhns, editors, *Distributed Artificial Intelligence 2*. Morgan Kaufmann,

1989.
52. A. Poggi. DAISY an Object-Oriented System for Distributed Artificial Intelligence. In M. Wooldridge and N.R. Jennings, editors, *Intelligent Agents. Proceedings of First International Workshop on Agent Theories, Architectures, and Languages (ATAL'94)*, number 890 in LNAI. Springer Verlag, August 1994.
53. A. Poggi. HOMAGE a Heterogeneous Object based Environment to Sevelop Multiagent Systems. In *Proceedings of the 29th Hawaii International Conference on System Sciences (HICSS-29)*, pages 282–289, Hawaii, 1996.
54. A. Rawston and A. 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.
55. J.S. Rosenschein and G. Zlotkin. *Rules of Encounter: Designing Conventions for Automated Negotiation among Computers.* MIT Press, 1994.
56. A. Sathi and M. Fox. Constraint-directed Negotiation of Resource Allocations. In L. Gasser and M. Huhns, editors, *Distributed Artificial Intelligence 2.* Morgan Kaufmann, 1989.
57. L. Schmutz. SAgA- An autonomous multi-agent platform for collective artificial intelligence. In *International Workshop on Decentralized Intelligent Multi-Agent Systems (DIMAS'95)*, pages 402–410, Krakow, Poland, November 1995.
58. S. Schubiger and O. Krone. Interactive Resource Sharing on the Web. In *Proceedings of Workshop on Distributed Computing on the WEB*, Rostock, Germany, June 22-23 1998.
59. J. Searl. *Speech Acts.* Cambridge University Press, 1969.
60. Shoham. Agent-Oriented Programming. *Artificial Intelligence*, 60 (1):51–92, 1993.
61. D.B. Skillicorn and D. Talia. *Programming Languages for Parallel Processing.* IEEE Computer Society Press, 1995.
62. R.G. Smith. The Contract net Protocol: High-level Communication and Control in a Distributed Problem Solver. *IEEE Trans on comput.*, 29(12), December 1980.
63. Sun Microsystems, Inc. *Java Space TM Specification, Revision 1.0*, March 1998.
64. K. Sycara. Multi-agent Compromise via Negotiation. In L. Gasser and M. Huhns, editors, *Distributed Artificial Intelligence 2.* Morgan Kaufmann, 1989.
65. S.R. Thomas. *PLACA, an Agent Oriented Programming Language.* PhD thesis, Computer Science Department, Stanford University, Stanford, 1993.
66. D. Weerasooriya, A. Rao, and K. Ramamohanarao. Design of a Concurrent Agent-Oriented Language. In M. Wooldridge and N.R. Jennings, editors, *Intelligent Agents. Proceedings of First International Workshop on Agent Theories, Architectures, and Languages (ATAL'94)*, number 890 in LNAI. Springer Verlag, August 1994.
67. K.J. Werkman. Knowledge-based Model of Negotiation using Shareable Perspectives. In *Proceedings of the 10th Int Workshop on DAI*, 1990.
68. J.E. White. Telescript technology: the Foundation for the electronic Markplace. Technical report, General Magic, Inc, Mountain view, CA, 1994.
69. M. Wooldridge. Agents as a Rorschach Test: A Response to Franklin and Graesser. In J.P. Muller, M.J. Wooldridge, and N.R. Jennings, editors, *Proceedings of ECAI'96 Workshop (ATAL). Intelligent Agents III. Agent Theories, Architectures, and Languages*, number 1193 in Lectures Notes in Artificial Intelligence, pages 47–48, 1996.

70. M. Wooldridge and N.R. Jennings. Agent Theories, Architectures, and Languages: a Survey. In M. Wooldridge and N.R. Jennings, editors, *Intelligent Agents*, number 890 in LNCS, pages 1–39. Springer Verlag, 1995.
71. T. Ziemke. Adaptive Behavior in autonomous agents. *To appear in Autonomous Agents, Adaptive Behaviors and Distributed Simulations' journal*, 1997.

This article was processed using the LaTeX macro package with LLNCS style