Coordinating Autonomous Entities with STL*

Oliver Krone, Fabrice Chantemargue, Thierry Dagaeff, Michael Schumacher
Computer Science Department, PAI group
University of Fribourg, CH-1700 Fribourg, Switzerland
http://www-iiuf.unifr.ch/pai

Abstract

This paper describes ECM, a new coordination model and STL its corresponding language. STL's power and expressiveness are shown through a distributed implementation of a generic autonomy-based multi-agent system, which is applied to a collective robotics simulation, thus demonstrating the appropriateness of STL for developing a generic coordination platform for autonomous agents.

Keywords: Coordination, Distributed Systems, Autonomous Agents, Collective Robotics.

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, 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 [30]), 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 distributed implementations burdensome. To study problems related to coordination, Malone and Crowston [25] 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 based on the coordination model ECM, which is a model for multi-grain distributed applications. STL is used so as to provide a coordination framework for distributed MAS made up of autonomous agents. It enables to describe the organizational structure or architecture of a MAS. It is conceived as a basis for the generic multi-agent platform CODA[1].

2 Coordination Theory, Models and Languages

Coordination can be defined as the process of *managing dependencies between activities* [25], or, in the field of Programming Languages, as the *process of building programs by gluing together active pieces* [10]. 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

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

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

key idea of the famous paper of Gelernter and Carriero [10] 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 [20]):

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*.

In [10] 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 [9] which is based on a *tuple space abstraction* as the underlying coordination model. An application of this model has been realized in PIRANHA [8] (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 [14] effort extends LINDA's tuple space onto the World-Wide-Web and BONITA [28] 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 [18], DARWIN [24], TOOLBUS [5]), *message passing paradigms* (COLA [17], ACTORS [1]), *object-oriented techniques* (OBJECTIVE LINDA [19], JAVASPACE [31]), *multi-set rewriting schemes* (BAUHAUS LINDA [11], GAMMA [4]) or *Linear Logic* (LINEAR OBJECTS [6]). A good overview on coordination issues, models and languages can be found in [27].

Our work takes inspiration from control-oriented models and tuple-based abstractions, and focuses on coordination for purpose of MAS distributed implementations.

3 Coordination using Encapsulation: ECM

ECM[2] 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 these blops active entities communicate anonymously through connections, established by the matching of the entities' communication interfaces.

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.

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

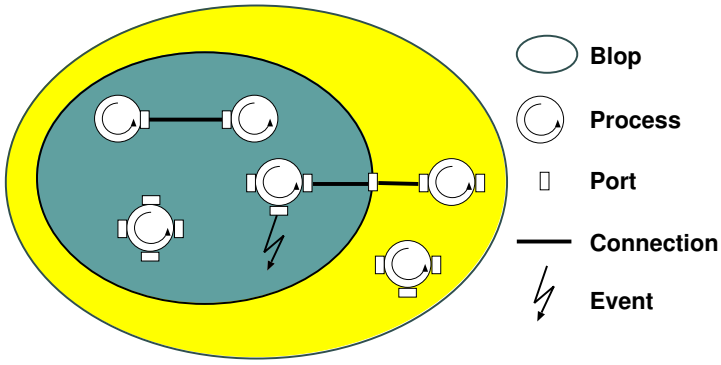

Figure 1: The Coordination Model of ECM.

3.1 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. 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. 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.

3.2 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.

3.3 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. Ports are characterized through a set of features from which the communication feature is mandatory and must be supported by all ECM realizations. The communication feature materializes the communication paradigm: it includes point-to-point stream communication (with classical message-passing semantics), closed group (with broadcast semantics) and blackboard communication. Additional port features specify e.g. the amount of other ports a port may connect to, see STL as an example.

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 have compatible values of features; (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 conditions (1)-(4) are fulfilled for two potentials in a blop, the connection between their corresponding ports is established and the potentials disappear.

3.4 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, i.e. 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.

3.5 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

```
blop world {
                    // Process definition
  process p1 {
    P2Pin port1 <"INPUT">;
    BB port2 <"BB">; // with its ports
    ...
    func p1;        // Thread entry point
  }
  ...               // More processes
  blop b1 {
    ...             // A new blop
  }
  ...               // More blops
  create process p1; // Create processes
  ...
  create blop b1();  // Create blops
  ...
}                   // End of blop world
```

Figure 2: Layout of a typical program written in STL.

checking is implementation dependent (see STL's event definition as an example of how to define event semantics on ports).

4 The Coordination Language STL

We designed and implemented a first language binding of the ECM model, called STL[3]. 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 [21] is based on PT-PVM [22], 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).

The ECM model is realized in an STL program whose general structure is outlined in Figure 2. Starting from the default blop `world`, a hierarchy of processes and blops can be defined, showing the hierarchical structure of the language at definition level.

4.1 STL's Specialities

In this Section we look at particularities for the instantiation of the ECM model in STL.

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

Blops

The name of a blop is used to create instances of a blop object. Blop objects can be placed onto a specific physical machine, or can be distributed onto a cluster of workstations. 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.

Figure 3 shows the definition of two blops (called `world` and `sieve`) in STL syntax, the line '`create blop sieve s()`' initializes the blop somewhere on the parallel machine. The statement could be annotated with a machine name to specify the actual workstation on which the blop should be initialized. The port definitions will be explained later.

```
blop world {
  ...
  blop sieve { // Declaration blop sieve
               // Two ports: types and names
    Group a <"CONNECTOR">;
    P2Pout b <"RESULT">;
  }
  ...
  create blop sieve s(); // Create blop
}
```

Figure 3: Blop declaration and invocation in STL.

Processes

ECM processes in STL can be activated from within the coordination language and in the computation language. 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 can be done in the body of an event or in the computation language directly. To some extent this is a trade-off regarding our goal to totally separate coordination and computation at code level. However, in order to preserve a high level of flexibility at application level, we allow these two possibilities.

Process termination is implicit: once the function which implements the process inside the computation language has terminated, the process disappears from the blop.

Figure 4 shows an example of an STL process type `worker` with two static ports `in` and `res` and a thread entry point `worker`; the syntax and semantics of the port definitions will be explained in the next section.

Ports and Connections

STL knows *static* ports as an interface of a process or blop defined in the coordination language, and *dynamic* ports which are created at runtime in the computation language. However, the type of a dynamic

Attribute	P2P	BB	Group	MyPort	Explanation
Communication	stream	blackboard	group	stream	Communication structure
Saturation	1	*	*	5	Amount of ports that may connect
Capacity	*	*	*	12	Capacity port: in data items
Synchronization	async	async	async	async	Semantics message passing model
Orientation	inout	inout	inout	in	Direction of data flow

Table 1: STL's built-in ports and a user defined port with corresponding port attribute values.

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

Figure 4: Process declaration and invocation in STL.

port, i.e, its features must be determined in the coordination language.

STL ports use a set of attributes as an implementation for ECM's port features. These attributes must be compatible in order to establish a connection between two ports. Table 1 gives an overview of the attributes of a port; combinations of attributes lead to port types.

STL provides the following built-in port types: point-to-point output ports, (`P2Pout`), point-to-point input ports (`P2Pin`), point-to-point bi-directional ports (`P2P`), groups (`Group`) and blackboards (`BB`). Variants of these types are possible and can be defined by the user.

`P2P:`
: The classical stream port. Two matched ports of this type result in a stream connection with the following semantics: Every send operation on such a port is non blocking, the port has an infinite storage capacity (in STL, infinity is symbolized by *), and matches to exactly one other port. The orientation attribute defines whether the port is an output port (`P2Pout`), an input port (`P2Pin`), or both (`P2P`).

`Group:`
: A set of `Group` ports form the group mechanism of STL. Ports of this type are gathered in a group and all message send operations are based on broadcast, that is, the message items will always be transfered to all members of the group. A closed group semantics is used.

`BB:`
: The `BB` stands for Blackboard and the resulting connection has a blackboard semantics as defined for ECM. In contrast to the previous port types, messages on the blackboard are now persistent objects and processes retrieve messages using a symbolic name and a tag.

Combinations of these basic port types are possible, for example to define a (`1:n`) point-to-point connection, the saturation attribute of a `P2P` port can be augmented to `n`, see Table 1 port `MyPort`.

Synchronous communication can be achieved by changing the type of message synchronization to synchronous, thus yielding in point-to-point synchronous communication. For `1:n` this means that the data producing process blocks until all the `n` processes have connected to the port, and every send operation returns only after all `n` processes have received the data item.

In STL the `synchronization` attribute overrides the `capacity` attribute, because synchronous communication implies a capacity of zero. However, asynchronous communication can be made a little bit less asynchronous by setting the capacity attribute to a value `n` to make sure that a process blocks after having sent n messages. Note that, the capacity attribute is a local relation between the process and its port: for asynchronous communication with a certain port capacity, it is only guaranteed that the message has been placed into the connection which does not necessarily mean that another process connected via a port to this connection has (or will) actually received the message.
Connections result in matched ports and are defined in accordance to the ECM model.

STL's Events

Events are triggered using a condition operation on a port. The event is handled by an event handler inside the blop.

Conditions related to ports of processes or blops determine when the event will be executed in the blop (for an overview on port conditions, see Table 2). Whether an event must be triggered or not will be checked by the system every time data flows through it or a process accesses it. Otherwise a condition like `isempty` would uninterruptedly trigger events for ports of processes, because at start-up of the process ports are empty.

Condition	Description
`unbound(port p)`	For `saturation` $\neq$ `*` the predicate `unbound()` returns `true` if the port has not yet matched to all its potential communication partners. For ports with `saturation=*`, the `unbound` predicate returns always `true`.
`accessed(port p)`	Equals `true` whenever the port has been accessed in general.
`isempty(port p)`	Checks whether the port has messages stored or not.
`isfull(port p)`	Returns `true` if the port's capacity has been reached.
`msg_handled(port p, int n)`, `less_msg_handled(port p, int n)`	Equals `true` if `n` messages, or less than `n` messages have been handled, respectively.

Table 2: Conditions on ports.

```
event new_worker() {
  create process worker new;
  when unbound(new.out) then new_worker();
}
                      // Process type worker
process worker {
  P2Pin in <"WORK">   // in port
  P2Pout out <"WORK"> // out port
  func worker;        // Thread entry pt
}
create process worker w;
  // Attach event to port
when unbound(w.out) then new_worker();
```

Figure 5: An example of event handling in STL.

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.

Very useful is the `unbound` condition on ports because it enables the construction of parallel software pipelines very elegantly. If we reconsider Figure 4 and extend it to Figure 5, we see the interaction of event conditions and ports in STL. First an event `new_worker` is declared. The event is attached to an `unbound` condition on the `out` port of the initial process `w`, denoted `w.out`. If process `w` either reads or writes data from/to its port `out`, the event `new_worker` is triggered because at that time there are no other ports to which `w.out` is currently bound, so `unbound(w.out)` returns TRUE. The event body of the event declaration of `new_worker` creates a new process of type worker, resulting in a new port signature or "potential" in the blop. The blop matches now the two ports (`w.out` and `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 the same condition (`unbound`) is attached to its port `new.out`.

This example illustrates the necessity of reinstalling explicitly event handlers so as to ensure a coordinated execution of the event handler body; in this case a process creation.

Primitives

STL is a separate language used in addition to a given computation language (in this case C), however the coordination entities must be accessed from within the computation language. Therefore, we implemented a C library to interact with the coordination facilities of STL. The set of primitives includes operations for creating dynamic ports, methods to transfer data from and to ports, and operations for process management; see [21] for a detailed specification.

4.2 STL Compiler and Runtime System

Figure 6 shows the basic building blocks of the STL programming environment in context with PT-PVM. A distributed application consists of two files: the coordination part (`app.stl`) and the computation part (`app-func.c`). The STL part will be parsed by the STL compiler to produce pure PT-PVM code. The final program will then be linked with the runtime libraries of STL and PT-PVM, the user supplied code, and the generated code of the STL compiler.

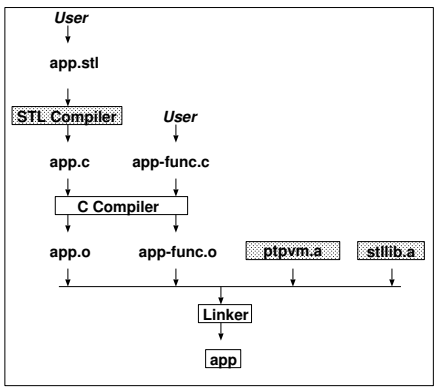

Figure 6: Programming environment of STL.

5 Coordination of Autonomous Agents in STL

One of our target with STL is the distributed implementation of a class of multi-agent systems.

The multi-agent methodology is a recent area of distributed artificial intelligence. A MAS is an organization given by a set of artificial entities acting in an environment. Focusing on collective behaviors, this methodology is aimed at studying and taking advantage of various forms of agent influences and interactions. It is widely used either in pure simulation of interacting entities (for instance in artificial life [23]) or in problem solving [26].

Definitions of MASs are general enough to address multiple domains which can be specified by the nature of the agents and the environment. We will proceed by composing a typical software MAS starting from problematics logically inducing a "distributed" approach and which could capitalize on self-organizing collective phenomena. One key concept in such approaches is *emergence*, that is the apparition of functional features at the level of the system as a whole. Our work is aimed at leading to robust solutions for applications in the frameworks of robotics and parallelism. For the design of our systems, we follow the "new AI" trend [7], [13]: our agents are *embodied* and *situated* into an environment. The primeval feature we attempt to embody into them is autonomy: the latter is believed to be a necessary condition for flexibility, scalability, adaptability and emergence.

In what follows, we will formally describe the class of MASs we attempt to implement on distributed architectures. Then we will motivate our project and discuss the difficulties encountered when distributing such MASs. We will eventually present the implementation with STL of a peculiar application belonging to our class of MASs.

5.1 A Generic Model for an Autonomous Agents' System

Our generic model is composed of an *Environment* and a set of *Agents*. The *Environment* encompasses a list of *Cells*, each one encapsulating a list of on-cell available *Objects* at a given time (objects to be manipulated by agents) and a list of connections with other cells, namely a *Neighborhood* which implicitly sets the topology. This way of encoding the environment allows the user to cope with any type of topology, be it regular or not, since the set of neighbors can be specified separately for every cell.

The general architecture of an agent is displayed on Figure 7. An agent possesses some sensors to perceive the world within which it moves, and some effectors to act in this world (embodiment). The implementation of the different modules presented on Figure 7, namely *Perception*, *State*, *Actions* and *Control algorithm* depends on the application and is under the user's responsibility. In order to reflect embodiment and situatedness, perception must be local: the agent perceives only the features of one cell, or a small subset of cells, at a given time. 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* or a *behavioral* autonomy [33]. 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, needs the freedom to have formed (learned or decided) its principles of behavior on its own (from its experience), at least in part. For instance, a very basic autonomy would consist of randomly choosing the type of action to take, a more sophisticated one would consist of implementing some learning capabilities, e.g. by using an adaptive neural network.

5.2 A Typical Application: *Gathering Agents*

Our class of MAS can support numerous applications. We illustrate with a simulation in the framework of mobile collective robotics. 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 8.

Our approach rests on a system integrating operationally autonomous agents, that is, each agent in the system acts freely on a cell (the agent decides which action to take according to its own control algorithm and local perception). Therefore, there is no master responsible for supervising the agents in the system, thus allowing it to be more flexible and fault tolerant. Agents have neither explicit coordination features for

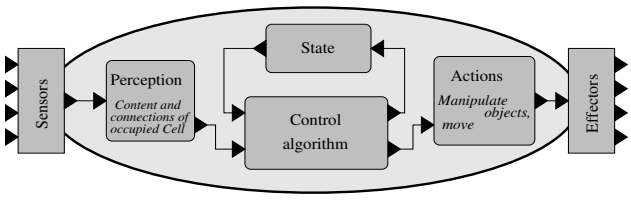

Figure 7: Architecture of an agent.

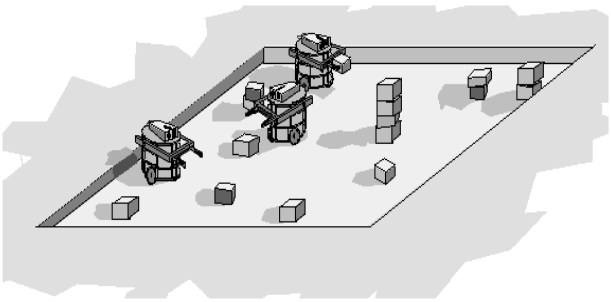

Figure 8: Collective robotics application: stacking objects.

detecting and managing antagonism situations (conflicts in their respective goals) nor communication tools for negotiation. In fact they communicate in an indirect way, that is, via their influences on the environment.

In our simulation, the environment is composed of a discrete two dimensional L cell-sided grid, a set of N objects and n mobile agents. An object can be located on a cell or carried by an agent. Under the given constraints, we implemented several variants for agent modules (details can be found in [12] and [16]). A simple control algorithm that can be used is as follows. Agents move randomly from a cell to a connected one. If an agent that does not carry an object comes to a cell containing NO objects, it will pick up one object with a probability given by $1/\mathtt{NO}^{\alpha}$, where $\alpha \geq 0$ is a constant; if an agent that carries an object comes to a cell containing some objects, it will systematically drop its object. If the cell is empty, nothing happens.

This simulation has been already serially implemented, exhibiting 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 that. Further details about this simulation and outcomes can be found in [16].

5.3 Coexistence and Distribution of Autonomous Agents

We would like to stress on the problem of implementing a MAS, such as the one described above, on a distributed architecture.

The instantiation of a MAS is most of the time serial. However that may be, because one draws inspiration from group of robots or living entities, it seems clear that the agents may run in parallel. At the level of abstraction of the MAS specification, the term parallelism simply conveys the notion *co-existence* of the autonomous agents. Hence, parallelism ideally underlies every conception of MAS and is thought to be implicitly taken into account in a serial implementation. Paradoxically, the projection of a MAS onto a distributed architecture turns out to be far from being obvious. We will not discuss here what are the fundamental origins of this difficulty; more details may be found in [15] (it concerns the different concepts of time in MAS and in distributed computing, respectively). We will illustrate the difference between concepts of coordination in both distributed computing and MAS.

The problem crystallizes around the different levels of what we understand when speaking about coordination.

In a conception stage (i.e. before any implementation), the notion of "coordination of agents" refers to a level of organization quite different from the one at programming level. We studied our MAS application (the gathering agents) in order to investigate some methods of cooperation between agents, namely emergent or self-organized cooperation, a sub-domain of coordination. In this stage, "cooperation between agents" deals with dependencies between agents as autonomous entities. The challenge is to find an appropriate trade-off between cooperation, the necessary fruitful coordination of inter-dependent entities, and their relative autonomy. At this stage, the envisaged cooperation methods must be described, in terms of the agents' architecture, their perception, their actual sensors and effectors. Thus, these methods have to be completely undertaken by their internal control algorithm: this is a result of the embodiment and situatedness prescriptions.

In an implementation stage, the notion of "coordination of agents" deals with the organization of the actual processes, or "pieces of software" (structures, objects, ...) which represent the agents at machine level. In a serial implementation, agents work in a round robin fashion in such a way that data consistency is preserved. Therefore, no further coordination problems occur. Thus, serial implementations do not to perturb the conceptual definition of coordination of the agents.

But in a distributed implementation, new problems arise, due to shared resources and data, synchronization and consistency concerns. Coordination models, in distributed computing, are aimed at providing solutions for these problems: they describe coordination media and tools, external to the agents, so as to deal with the consequences of the spatial distribution of the supporting processes. For example, the coordination medium, such as ports and connections in ECM, is the substratum in which coordinated entities are embedded for what concerns their "coordi-

nation dimension". This substratum should not be confused with the agents' environment described at MAS level. It reflects the distributed supporting architecture. We find here again the notion of orthogonality: the coordination medium is orthogonal to the model of the agents and their environment. If we take the separation of concern into account, this means that the coordination introduced at the conceptual level, which has to be implemented in the control algorithm module of the agents, belongs to the computational part, and has to be implemented in the computational language, whereas the coordination of the supporting processes has to be implemented in the coordination language.

Nevertheless, the question is now to determine to which extent this separation between computation (including agents' conceptual coordination) and coordination (of distributed processes) is possible. In other words, to which extent coordination methods of distributed processes do not interfere with coordination methods of agents as specified at MAS level; and what kind of coordination media and tools may provide coordination means compatible with the agent architecture, their autonomy and locality prescriptions.

Usual platforms that enable distributed computing do not belong to MAS domain. Distributed implementations of a MAS through existing languages would give rise, if no precautions are taken, to a hybrid system which realizes an improper junction between the two levels of definition of coordination. In each case, the agent processes are coordinated or synchronized at a rate and by means out of the conceptual definition of the MAS, but that the chosen language provides and compels to use in order to manage its processes. Because the processes are not designed with the aim of representing autonomous agents, the resulting system may exhibit characteristics which are not the image of a property of the MAS itself. One of our goals is to understand and prescribe what precautions are to be taken, and to develop a platform that makes distributed implementations of our MAS class an easier task.

For this, we start with an implementation of our MAS class with STL.

5.4 Constraints for a Distributed Implementation

Our very aim is to be able to express our autonomy-based multi-agent model on a distributed architecture in the most natural way which preserve autonomy and identity of the agents. We attempt to use STL in order to distribute our system of gathering agents. The problematics sketched here above is well reflected when we try to distribute the environment itself on several processes (machines). The only purpose of this division of the environment (for instance 4 blocks of $(L/2)^2$ cells each) is to take advantage of a given distributed architecture. But it clearly necessitates means in addition to coordination mechanisms described at MAS level: a mechanism is needed 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). Moreover, we will need another type of mechanism in order to cope with data consistency (e.g. updating the number of objects on a cell). These mechanisms should not alter every agent's autonomy and behavior: we will have to dismiss any unnecessary dependency.

5.5 Implementation in STL

The *Environment* is a torus grid, in which every cell has four neighbors (four connectivity). Note that using a four connectivity (against an eight connectivity) basically does not change anything except that it slightly alleviates the implementation. Agents comply rigorously with the model previously introduced in Figure 7. They sense the environment through their sensors and act upon their perception at once.

To take advantage of distributed systems, the *Environment* is split into sub-environments, each of which being encapsulated in a blop, as depicted in Figure 9, thus providing an independent functioning between sub-environments (and hence between agents roaming in different sub-environments). Note that blops have to be arranged so as to preserve the topology of the sub-environments they implement.

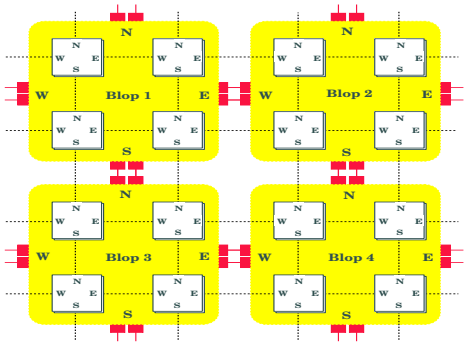

Figure 9: Environment split up among 4 blops.

For our implementation new port types have been introduced, namely `P2P_Nin` and `P2P_Nbi`, which are respectively variants of `P2Pin` and `P2P`, for which the `saturation` attribute is set to infinity (see Figure 10).

```
port P2P_Nin {
  communication = stream;
  orientation = in;
  capacity = *;
  saturation = *;
  synchronization = async;
}
port P2P_Nbi {
  communication = stream;
  orientation = inout;
  capacity = *;
  saturation = *;
  synchronization = async;
}
```

Figure 10: User-defined port types.

Global Structure

The meta-blop *world* is composed of an *init* process, in charge of the global initialization of the system, and a set of N pre-defined blops (called *bx* with x ranging from *1* to *N*), each of which encapsulating and handling a sub-environment. Figure 11 gives a graphical overview of the organization within a blop *bx* in the meta-blop *world*. Note that on this figure, only one blop *bx* is represented so as to avoid to overload the picture. In the case of an application with multiple blops *bx*, there should be some connections between the *init* process and all the blops, as well as some connections between north ports of top blops with south ports of bottom blops and east ports of east blops with west ports of west blops have been intentionally dropped

The *init* process has four static ports for every blop to be initialized: three of type `P2Pout` (*init_NbAgts*, *cre_Agts* and *cre_SubEnv*) and one of type `P2P` (*eot*). The rôle of the *init* process is threefold: first, to create through its *init_NbAgts* and *cre_Agts* ports the initial agents within every blop; second, to set up through its *cre_SubEnv* port the sub-environment (size, number of objects and position of the objects on the cells) of every blop; third, to collect the result of an experiment, to signal the end of an experiment, and to properly shutdown the system through the *eot* port.

Blops *bx*: Figure 13 (see Appendix) shows how the implementation of the application depicted on Figure 11 looks like in the STL coordination language, in the case of four blops *bx*, namely *b1*, *b2*, *b3* and *b4*. Figure 12 (see Appendix) presents the declaration and instantiation of all the processes belonging to a blop *bx*. Two types of processes may be distinguished: processes that are purpose-built for a distributed implementation of the multi-agent application (they enable a distributed implementation), namely *initAgent* and *taxi*, and processes that are actually peculiar to the multi-agent application, viz. *subEnv* and *agent* processes.

Ports of a Blop *bx*: Each blop has twelve static ports: four `P2Pout` *outflowing direction* ports (*north_o, south_o, west_o, east_o*) and four `P2Pin` *inflowing direction* ports (*north_i, south_i, west_i, east_i*), which are gateway ports enabling agent migration across blops; three `P2Pin` ports, namely *b_NbAgts*, *b_Agents* and *b_SubEnv* used for the creation of the initial agents (actually realized in the *initAgent* process) and for an appropriate setup of the sub-environment (achieved in the *subEnv* process); and a `P2P` port (*b_eot*) used to forward to the *init* process the result of an experiment and to indicate the end of an experiment.

For the time being, the topology between blops is set in a static manner, by creating the ports with appropriate names (see Figure 13 of the Appendix). The four *inflowing direction* ports of a blop match with a port of its inner process *initAgent*. The four *outflowing direction* ports of a blop match with ports of its inner process *taxi*.

***initAgent* Process, *new_agt* Event**: The *initAgent* process is responsible for the creation of the agents. It has four static ports: *nb_Agts* of type `P2Pin`, *newArrival* of type `P2P_Nin`, *location* of type `P2P` and *init* of type `P2Pout`. At the outset of the experiment, the *initAgent* process through its *nb_Agts* port will be informed by the *init* process on the number of agents to be created in the present blop. The *initAgent* process will then loop on its *newArrival* port so as to receive the identifiers of the agents to be created. As soon as a value comes to this port, the *new_agt* event (see Figure 12 of the Appendix) is triggered and it will create a new *agent* process. In the meantime, the *initAgent* process will draw randomly for every agent's identifier an agent's position. The *location* port enables the *initAgent* process to communicate with the *subEnv* process, so as to have a better control on the position of an agent with regard to other agents' and objects' positions, e.g. to ensure that at the outset no more than one agent can reside on an empty cell. The *initAgent* process will then write on its *init* port some values for the agent just created. The latter, through its *creation* port will read the information that was previously written on the *init* port of the *initAgent* process. Values that are transmitted feature for instance the position of the agent and its *state*.

Note that the *newArrival* port is connected to all *inflowing direction* ports of the blop within which it resides, thus enabling to deal with migrating agents across blops in the course of an experiment, by the same event mechanisms as described above. The *lo-*

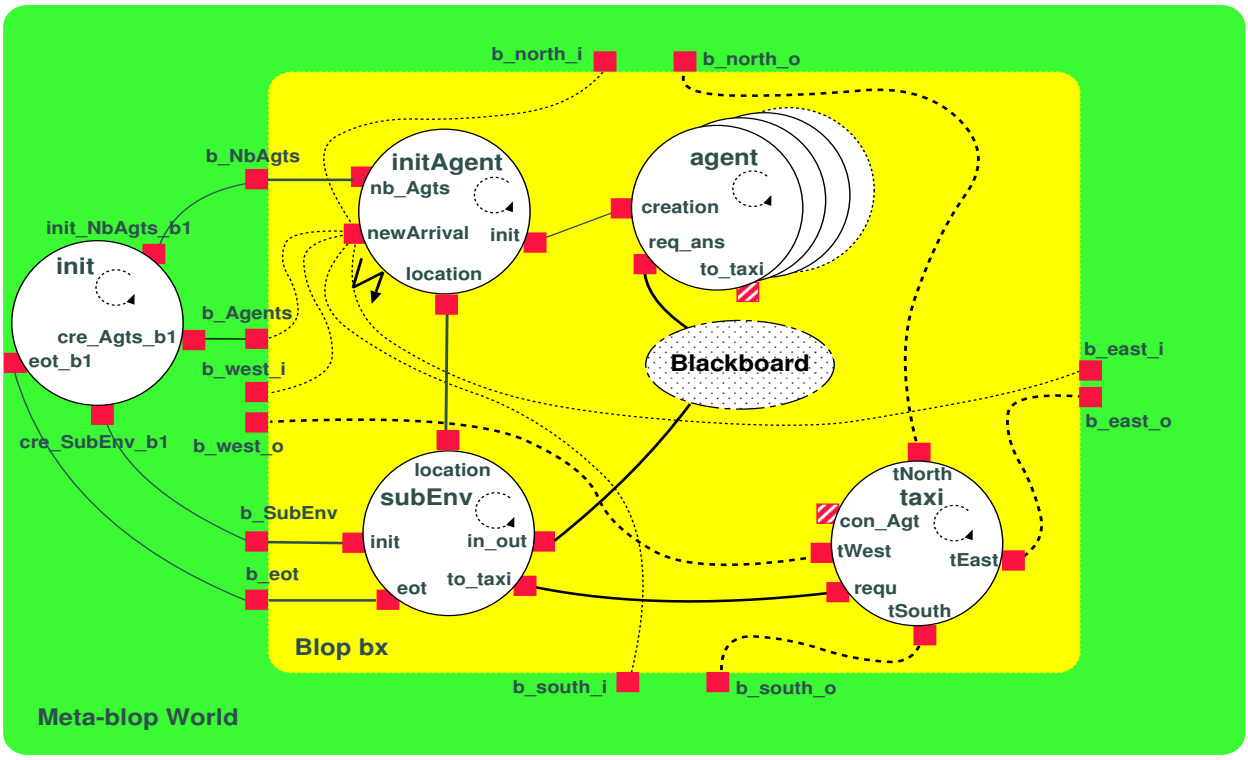

Figure 11: *init* process and a single blop *bx*: solid and dotted lines are introduced just for a purpose of visualization.

cation port is very useful at this point, because the *initAgent* process can already check with the *subEnv* process whether the position the agent intends to move to is permitted or not. In case it is not, the *initAgent* process will have to draw randomly a new position in the neighborhood of the position the agent intended to go.

***agent* Process**: This process has two static ports (*req_ans* of type `BB` and *creation* of type `P2Pin`) plus *to_taxi* a dynamic `P2Pout` port. As already stated, this process reads on its *creation* port some values (its position and its *state*). All *req_ans* ports of the agents are connected to a *Blackboard*, through which agents will sense their environment (*perception*) and act (*action*) into it, by performing LINDA-like *in*/*out* operations with appropriate messages. The type of action an agent can take depends on the type of *control Algorithm* implemented within the agent (see the architecture of an agent on Figure 7). The *to_taxi* port is used to communicate dynamically with the *taxi* process in case of migration: the position and *state* of the agent are indeed copied to the *taxi* process. The decision of migrating is always taken by the *subEnv* process.

***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, which will cross the border of a sub-environment. It has a static *in_out* port (of type `BB`) connected to the *Blackboard* and a static `P2Pout` port *to_taxi* connected to the *taxi process*. Once initialized through its *init* `P2Pin` port, the *subEnv* process builds the sub-environment. By performing *in*/*out* operations with appropriate tuples, the *subEnv* process will process the requests of the agents (e.g. number of objects on a given cell, move to next cell) and reply to their requests (e.g. x objects on a given cell, move allowed and registered). When the move of an agent will lead to cross the border (cell located in another blop), 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 direction the agent has to take will be transmitted, so that the *taxi* process can know which port to write to). The *location* `P2P` port is used to communicate to the *initAgent* process further to its request on the position of an agent with regard to other agents' and objects' positions.

The *taxi* Process: The *taxi* process is responsible for migrating agents across blops. It has four static *direction* ports (of type `P2Pout`), which are connected to the four *outflowing direction* ports of the blop within which it stands. When this process receives on its static `P2Pin` port *requ* the direction towards where an agent has to migrate, it will create

a dynamic `P2Pin` 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 (intended position plus *state*). These values will then be written on the port corresponding to the direction to take and will be transferred to the *newArrival* 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 Discussion

6.1 STL 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 an explicit hierarchical model.

- 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 and processes, 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 themselves. In fact, the latter can create new processes and new ports, thus yielding to communication topology changes.

ECM and STL present similarities with several other coordination models and languages like LINDA [9], DARWIN [24] or CONCOORD [18]. We mention few other specific characteristics of our work. Like several further developments of the LINDA model (for instance Objective Linda [19]), ECM uses a hierarchical multiple coordination space model, in contrast to the single flat tuple space of the original LINDA. Processes get started through an event in a blop, or automatically upon initialization of a blop, or through a creation operation by another process; LINDA uses one single mechanism: `eval()`. Processes do not execute in a medium which is used to transfer data. In order to communicate, they do not have references to other processes or to ports belonging to other entities; they communicate anonymously through their ports.

6.2 The agent language STL++, a new instantiation of the ECM model

Besides the advantage of a better overview of coordination duties, it however turned out that the separation of *code* (as in STL) 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 another instantiation of ECM, namely the coordination language STL++. This new language binding implements ECM by enriching a given object oriented language (C++ in our case) with coordination primitives, offering high dynamical properties. An STL++ application is then a set of classes inheriting from the basic classes of the STL++ library. STL++ aims at giving basic constructs for the implementation of generic multi-agent platforms, thus being an agent language [32]. A thorough description of STL++ can be found in [29].

7 Conclusion

In this paper, we presented the ECM coordination model and STL, its language binding. We built a

first STL-based prototype on top of the existing PT-PVM platform [22]. An implementation of a classical collective robotics simulation illustrated the power of STL and demonstrated its appropriateness for coordinating a class of autonomous agents, whose most critical constraint is the preservation of autonomy by dismissing coordination mechanisms exclusively embedded for purpose of implementation (unnecessary dependencies).

As far as the development of a platform for multi-agent programming is concerned, STL can be seen as a first starting point. STL already includes mechanisms which are 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 notion of ports avoiding any additional coordinator process; and (3) in despite of (2) the notion of blop hierarchy which in our case allows us to represent the encapsulation of the environment and the agents.

The STL coordination model is still to be extended in order to encompass as many generic coordination patterns as possible, yielding in STL skeletons at disposal for general purpose implementations. Future works will consist in: (1) improving the model, such as introducing new user-defined attributes for ports, dynamic ports for blops, data typing for port types, refining sub-typing of ports, supporting multiple names for ports, and (2) developing a graphical user interface to facilitate the specification of the coordination part of a distributed application.

There are two major outcomes to this work. First, as autonomous agents' systems are aimed at addressing problems which are naturally distributed, our coordination platform provides a user the possibility to have an actual distributed implementation and therefore to benefit from the numerous advantages of distributed systems, so that this work is 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 model.

Acknowledgements

We are grateful to André Horstmann and Christian Wettstein for their valuable work, which consisted in realizing parts of the STL platform.

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, *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] J.A. Bergstra and P. Klint. The ToolBus Coordination Architecture. In Paolo Ciancarini and Chris Hankin, editors, *First International Conference on Coordination Models, Languages and Applications*, number 1061 in LNCS. Springer Verlag, April 1996.

[6] 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.

[7] R.A. Brooks. Intelligence without Reason. In *Proceedings of IJCAI-91*, Sydney, Australia, 1991.

[8] N. Carriero, E. Freeman, D. Gelernter, and D. Kaminsky. Adaptive Parallelism and Piranha. *IEEE Computer*, 28(1), January 1995.

[9] N. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, 1989.

[10] N. Carriero and D. Gelernter. Coordination Languages and Their Significance. *Communications of the ACM*, 35(2):97–107, February 1992.

[11] 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*, number 924 LNCS, 1995. Springer Verlag.

[12] F. Chantemargue, T. Dagaeff, M. Schumacher, and B. Hirsbrunner. Coopération implicite et

performance. In *Sixth Symposium on Cognitive Sciences (ARC)*, Villeneuve d'Ascq, France, December 10-12 1996.

[13] F. Chantemargue, O. Krone, M. Schumacher, T. Dagaeff, and B. Hirsbrunner. Autonomous Agents: from Concepts to Implementation. In *Fourteenth European Meeting on Cybernetics and Systems Research (EMCSR'98)*, volume 2, pages 731–736, Vienna, Austria, April 14-17 1998.

[14] P. Ciancarini, A. Knoche, R. Tolksdorf, and Fabio Vitali. PageSpace: An Architecture to Coordinate Distributed Applications on the Web. In *Fifth International World Wide Web Conference*, volume 28 of *Computer Networks and ISDN Systems*, 1996.

[15] T. Dagaeff and F. Chantemargue. Performance of Autonomy-based Systems: Tuning Emergent Cooperation. Technical Report 98-20, Computer Science Department, University of Fribourg, Fribourg, Switzerland, October 1998.

[16] T. Dagaeff, F. Chantemargue, and B. Hirsbrunner. Emergence-based Cooperation in a Multi-Agent System. In *Second European Conference on Cognitive Science (ECCS'97)*, pages 91–96, Manchester, U.K., April 9-11 1997.

[17] B. Hirsbrunner, M. Aguilar, and O. Krone. CoLa: A Coordination Language for Massive Parallelism. In *ACM Symposium on Principles of Distributed Computing (PODC)*, Los Angeles, California, August 14–17 1994.

[18] 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.

[19] 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.

[20] T. Kielmann and G. Wirtz. Coordination Requirements for Open Distributed Systems. In *Proceedings of PARCO'95.* Elsevier, 1996.

[21] O. Krone. *STL and Pt-PVM: Concepts and Tools for Coordination of Multi-threaded Applications.* PhD thesis, University of Fribourg, 1997.

[22] O. Krone, B. Hirsbrunner, and V.S. Sunderam. PT-PVM+: A Portable Platform for Multithreaded Coordination Languages. *Calculateurs Parallèles*, 8(2):167–182, 1996.

[23] C. Langton. *Artificial Life.* Addison-Wesley, 1989.

[24] J. Magee, N.Dulay, and J. Kramer. Structuring parallel and distributed programs. *Software Engineering Journal*, pages 73–82, March 1993.

[25] T.W. Malone and K. Crowston. The Interdisciplinary Study of Coordination. *ACM Computing Surveys*, 26(1):87–119, March 1994.

[26] J.P. Muller. *The Design of Intelligent Agents: A Layered Approach.* Number 1177 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1996.

[27] 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.

[28] A. Rawston and A. Wood. BONITA: A Set of Tuple Space primitives for Distributed Coordination. In R. H. Sprague Jr., editor, *30th Hawaii International Conference on System Sciences*, volume 1, Wailea, Hawaii, 1997. IEEE. Minitrack on Coordination Languages, Systems and Applications.

[29] M. Schumacher, F. Chantemargue, O. Krone, and B. Hirsbrunner. STL++: A Coordination Language for Autonomy-based Multi-Agent Systems. Technical report, Computer Science Department, University of Fribourg, Fribourg, Switzerland, March 1998.

[30] D.B. Skillicorn and D. Talia. *Programming Languages for Parallel Processing.* IEEE Computer Society Press, 1995.

[31] Sun Microsystems, Inc. *Java Space TM Specification, Revision 1.0*, March 1998.

[32] 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.

[33] T. Ziemke. Adaptive Behavior in autonomous agents. *Autonomous Agents, Adaptive Behaviors and Distributed Simulations' journal*, 1997.

Appendix: STL Code Examples

Figure 13 shows how the implementation of our MAS application looks like in the STL coordination language, in the case of four blops *bx*, namely *b1*, *b2*, *b3* and *b4*. Figure 12 presents the declaration and instantiation of all the processes belonging to a blop *bx*.

```
process InitAgent {
    // to communicate with blop bx
  P2Pin nb_Agts <"BLOP_NB_AGTS_BX">;
  P2P_Nin newArrival <"BLOP_*_AGENTS_B*">;
    // to communicate with agent processes
  P2P_Nbi init <"AGENT">;
    // to communicate with subEnv process
  P2P location <"CHECK_LOCATION">;
  func fInitAgent;
}
create process InitAgent new_IA;
process SubEnv {
    // to communicate with blop bx
  P2Pin init <"BLOP_INIT_SUBENV_BX">;
  P2Pout eot <"BLOP_END_OF_TASK_BX">;
    // to communicate with agent processes
  BB in_out <"COMM_WITH_SUBENV">;
    // to communicate with taxi process
  P2Pout to_taxi <"INFO_TAXI_ENV">;
    // to communicate with initAgent process
  P2P location <"CHECK_LOCATION">;
  func fSubEnv;
}
create process SubEnv new_SE;
process Agent {
    // to communicate with initAgent process
  P2P creation <"AGENT">;
    // to communicate with subEnv process
  BB req_ans <"COMM_WITH_SUBENV">;
    // to communicate with taxi process
  P2Pout to_taxi <"INFO_TAXI_AGENT">;
  func fAgent;
}
event new_agt() {
    // create agent process through event
  create process Agent n;
  when accessed(new_IA.newArrival) then new_agt();
}
when accessed(new_IA.newArrival) then new_agt();
process Taxi {
    // to communicate with subEnv process
  P2Pin req <"INFO_TAXI_ENV">;
    // to communicate with agent processes
  P2Pin con_agt <"INFO_TAXI_AGENT">;
    // to communicate with blop bx
  P2Pout tNorth <"BLOP_No*">;
  P2Pout tEast <"BLOP_Eo*">;
  P2Pout tSouth <"BLOP_So*">;
  P2Pout tWest <"BLOP_Wo*">;
  func fTaxi;
}
create process Taxi new_T;
```

Figure 12: Inner part of a blop *bx* in STL: process declaration and instantiation. The star (*) represents a wild-card for matching.

```
blop world {
  process Init {
      // to communicate with blop b1
    P2Pout init_Nb_Agts_b1 <"BLOP_NB_AGTS_B1">;
    P2Pout cre_Agts_b1 <"BLOP_INIT_AGENTS_B1">;
    P2Pout cre_SubEnv_b1 <"BLOP_INIT_SUBENV_B1">;
    P2Pin eot_b1 <"BLOP_END_OF_TASK_B1">;
      // to communicate with blop b2
    P2Pout init_Nb_Agts_b2 <"BLOP_NB_AGTS_B2">;
    P2Pout cre_Agts_b2 <"BLOP_INIT_AGENTS_B2">;
    P2Pout cre_SubEnv_b2 <"BLOP_INIT_SUBENV_B2">;
    P2Pin eot_b2 <"BLOP_END_OF_TASK_B2">;
      // to communicate with blop b3
    P2Pout init_Nb_Agts_b3 <"BLOP_NB_AGTS_B3">;
    P2Pout cre_Agts_b3 <"BLOP_INIT_AGENTS_B3">;
    P2Pout cre_SubEnv_b3 <"BLOP_INIT_SUBENV_B3">;
    P2Pin eot_b4 <"BLOP_END_OF_TASK_B3">;
      // to communicate with blop b4
    P2Pout init_Nb_Agts_b4 <"BLOP_NB_AGTS_B4">;
    P2Pout cre_Agts_b4 <"BLOP_INIT_AGENTS_B4">;
    P2Pout cre_SubEnv_b4 <"BLOP_INIT_SUBENV_B4">;
    P2Pin eot_b3 <"BLOP_END_OF_TASK_B4">;
    func fInit;
  }
  blop b1 {
      // to communicate with initAgent process
    P2Pin b_Nb_Agts <"BLOP_NB_AGTS_B1">;
    P2Pin b_Agents <"BLOP_INIT_AGENTS_B1">;
      // to communicate with subEnv process
    P2Pin b_SubEnv <"BLOP_INIT_SUBENV_B1">;
    P2Pout b_eot <"BLOP_END_OF_TASK_B1">;
      // to communicate with south ports of
         blop just above
    P2Pin b_north_i <"BLOP_NiSo_AGENTS_B1B3">;
    P2Pout b_north_o <"BLOP_NoSi_AGENTS_B1B3">;
      // to communicate with west ports of
         blop just on west hand side
    P2Pin b_east_i <"BLOP_EiWo_AGENTS_B1B2">;
    P2Pout b_east_o <"BLOP_EoWi_AGENTS_B1B2">;
      // to communicate with north ports of
         blop just underneath
    P2Pin b_south_i <"BLOP_SiNo_AGENTS_B1B3">;
    P2Pout b_south_o <"BLOP_SoNi_AGENTS_B1B3">;
      // to communicate with east ports of
         blop just on east hand side
    P2Pin b_west_i <"BLOP_WiEo_AGENTS_B1B2">;
    P2Pout b_west_o <"BLOP_WoEi_AGENTS_B1B2">;
    ... Decl. and instantiation of processes ...
  }
  create blop b1 new_b1();
  blop b2 {
    ... Declaration and
    instantiation of processes ...
  }
  create blop b2 new_b2();
  blop b3 {
    ... Declaration and
    instantiation of processes ...
  }
  create blop b3 new_b3();
  blop b4 {
    ... Declaration and
    instantiation of processes ...
  }
  create blop b4 new_b4();
  create process Init new_Init;
}
```

Figure 13: Implementation of meta-blop world and four blops *b1*, *b2*, *b3*, *b4* in STL.