Coordinating Autonomous Entities*

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

Abstract

This paper describes STL, a new coordination model and corresponding language. STL's power and expressiveness are shown through a preliminary 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 models such as *Multi-Agent Systems* are oriented towards interactions, collaborative phenomena and autonomy. We will focus on a generic class of autonomous agents, from which we draw a typical application related to collective robotics, in order to validate our coordination approach.

Today's state of the art parallel programming models used for implementing general purpose distributed applications 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 a distributed implementation of autonomy-based multi-agent systems, our target application, a burdensome task. To study problems related to coordination, Malone [19] introduced a new theory called *Coordination Theory* aimed at defining such a "glue". Principles developed in this theory draw their inspiration not only from computer science, but from other disciplines, such as organization theory, operations research, economics, linguistics, biology and psychology.

When coordination theory is applied to computer science, the key issue is *managing dependencies* among activities. To formalize and better describe these interdependencies it is necessary to separate the two essential parts of a parallel application, namely *computation* and *coordination* [7]. These parts usually interfere with each other, so that distributed applications are hard to understand. The research in this area has focused on the definition of several coordination models and corresponding coordination languages.

A coordination language is the "*linguistic embodiment of a coordination model*" [7] and should be defined orthogonally to a computation language. The most prominent representative of this class of new languages is Linda [11] which is based on a tuple space abstraction as the underlying coordination model. An application of this model has been realized in Piranha [6] (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 [9] effort extends Linda's tuple space onto the World-Wide-Web and BONITA [20] addresses performance issues for the implementation of Linda's `in` and `out` primitives. Other languages and models are based on a control oriented approach [2], [18], message passing paradigms [12], [1], object-oriented techniques [14], multi-set rewriting schemes [8], [4] or Linear

*This work is financially supported by the Swiss National Foundation for Scientific Research, grants 21-43558.95 and 21-47262.96

Logic [5].

The rest of this paper is organized as follows. Section two describes in detail STL, our coordination model, and appropriate coordination language. Section three is devoted to an illustration of the power and appropriateness of STL through a preliminary implementation of a generic autonomy-based multi-agent system, applied to a mobile collective robotics simulation. In the last Section, we draw some conclusions about this work and outline future works.

2 Coordination Model of STL

STL[1] materializes the separation of concern as it uses a separate language exclusively reserved for coordination purposes and provides primitives which are used in the computation language to interact with the entities to be coordinated. It shares many characteristics with the IWIM [3] model of coordination like ConCoord [13] or MANIFOLD [2].

The coordination model of STL comprehends five building blocks which will now be introduced gradually:

1. *Processes*, as a representation of active entities;
2. *Blops*, as an abstraction and modularization mechanism for processes and ports;
3. *Ports*, as the interface of processes/blops to the external world;
4. *Events*, a mechanism to react to dynamic state changes;
5. *Connections*, as a representation of connected ports.

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

- The *Coordination Entities* of STL are the processes of the distributed application;
- There are two types of *Coordination Media* in STL: events, ports, and connections which enable coordination, and blops, the repository in which coordination takes place;
- The *Coordination Laws* are defined through the semantics of the *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.

Figure 1 gives a first overview of the programming metaphor on which STL is based. An STL application 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.

The reminder of this Section is devoted to a description of each element.

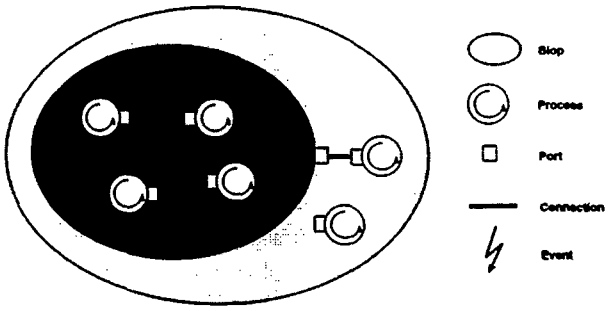

Figure 1: The Coordination Model of STL.

2.1 Blop

A blop is an abstraction for an agglomeration of objects to be coordinated and serves as a separate name space for port objects, processes, and subordinated blops as well as an encapsulation mechanism for events.

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

The creation of a blop is handled in the same way as the creation of processes (see 2.2). It includes the initialization of all static processes/blops and ports defined for this blop and subordinated blops.

2.2 Processes

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

[1] Simple Thread Language. STL is part of the CoLMA (Coordination Language for Multi-threaded Applications) effort of the University of Fribourg, which aims at developing tools for coordination of multi-threaded applications on a cluster of workstations.

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

Processes 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.

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

2.3 Ports

Ports are the interface of processes and blops to establish connections to other processes/blops, i.e. communication in STL is handled via a connection and therefore over ports. A port has a *name* and a set of well defined *attributes* and belongs either to a process or a blop. The port name and its attributes are referred to as the port's *signature*. The combination of port attributes results in a port type. We distinguish static and dynamic ports. Both static and dynamic ports are represented in a blop by port signatures. A *static* port is an interface of a process or blop defined in the coordination language, whereas a *dynamic* port will be created dynamically at runtime in the computation language. However, the type of the dynamic port, i.e, its attributes must be determined in the coordination language.

2.3.1 Port Attributes

Pairs of ports must comply with a set of attributes (see Table 1 for an overview) in order to match. As an example we explain the *communication* attribute in more detail. We provide the three classical communication paradigms: point-to-point stream communication, group and blackboard communication.

For point-to-point stream communication, the data distribution scheme is different. Processes communicate in a stream using the classical message passing semantics. Messages are tagged and can be received only once.

For group communication, a set of matched ports forms a closed group in which data will be transferred to all members of the group via a broadcast operation. The group is closed because a process must be member of the group in order to be able to send/receive data to and from the group. Each single process connected to such a group receives the information in the same way as in point-to-point streams.

For blackboard communication, the information can be retrieved from the port in a sequence defined by the process, and information can be retrieved more than once. Processes can put information onto this blackboard, read from it, or remove messages from it.

2.3.2 Basic Port Types

The combination of different port attributes yields to different port types. We have identified the following major port types: point-to-point output port, (`P2P->`), point-to-point input port (`P2P<-`), point-to-point bi-directional port (`P2P<->`), groups (`Group`) and blackboards (`BB`). Variants of these types are possible and can be defined by the user by modifying the port's attributes of Table 1.

`P2P`:
: The classical stream ports. 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, a receive call blocks the calling process until data is available and the port has an ∞ storage capacity, and matches to exactly one other port. The orientation attribute defines whether the port is an output port (`P2P->`), an input port (`P2P<-`), or a bidirectional port (`P2P<->`).

`Group`:
: A set of `Group` ports forms 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 transferred to all members of the group. A closed group semantics is used, processes must be member of the group in order to distribute messages in it.

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

  This multiple blackboard model provides a certain degree of privacy and encapsulation for communicating processes which is not present in the original Linda model. In order to access the information, the process must specify both, a specific port (to get access to the blackboard) and the name and tag of the data item to retrieve. Modularity is supported in so far as the blackboards serve as a private name space for a group of pro-

Attribute	Example	Explanation
Communication	`blackboard, stream, group`	Communication structure
Saturation	`saturation=7`	Seven other ports may connect; default: 1
Capacity	`capacity=5`	Capacity of a port: 5 data items; default: ∞
Msg. Synchronization	`synchron, asynchron`	Semantics of message passing model
Orientation	`in, out, inout`	Direction of data flow

Table 1: Attributes of a port.

cesses which form a software module. Therefore, each module can independently use the same message tuples without interfering with other modules.

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

2.3.3 Variations of the Basic Port Types

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

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 have received the data item.

One can say that the type of `Msg. Synchronization` is "stronger" than the `Capacity` attribute, because synchronous communication implies a capacity of zero. On the other hand, asynchronous communication can be made a little bit less asynchronous by setting the capacity attribute to a certain value `n` to make sure that at least after `n` messages the process blocks. However the capacity attribute is a local relation between the process and its port.

2.3.4 Port Matching

The matching of ports is defined as a relation between port signatures. It is not a static relation which can be determined at compilation time, but depends on the current state of the port relative to its attributes. In other words, although the signatures of two ports may match at compile time they do not match at runtime because, e.g. the number of communication partners which may be able to connect to this port is limited (through the `saturation` attribute).

There are five conditions that must be fulfilled in order for two ports to match: (1) both use the same communication attribute, (2) both have the same name, (3) both ports must not be saturated, (4) both belong to the same level of abstraction, i.e., are visible within the same hierarchy of blops, and (5) 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 current blop where it is embedded. If two compatible potentials exist in the blop, and if the conditions (1)-(5) are fulfilled, the connection between the corresponding ports is established and the potentials disappear. The notion of compatible potentials introduces a subtype relation on port types, thus permitting the matching of ports whose attribute values are not necessarily identical.

For blops the scheme works analogously. As for processes the port represents the blop's interface to the outside world. For blops this means that "one side" of the port is visible inside the current blop, and "the other side" is visible outside, that is, in the blop where the blop is embedded. The blop creates with its port signatures a potential in two encapsulated environments.

Whether two potentials match or not depends on the communication structure:

- *Directed stream point-to-point communication.* Due to the nature of this communication structure, a "negative" and a corresponding "positive" potential must exists in the current blop to form a connection. The negative potential represents an input port, a positive potential symbolizes an output port. To avoid that a port may consume all potentials in a blop we define that a port never matches twice to the same potential.

  For blops we define that for an input port, the negative potential is created in the surrounding blop and the positive potential is created in the current blop. For output ports the reverse mechanism is applied.

- *Bi-directional communication and groups.* The mechanism works analogously to directed stream point-to-point communication with the difference that neutral potentials are created. The communication partners are identified by the

port name and attributes. For groups this means that all processes using a compatible group port type are grouped together. For blops this means that the group which is otherwise only locally visible will be exported to the surrounding blop.

- *Blackboards.* The potentials for such a communication structure are also neutral and always present in the blop because the communication partner is fully determined through the port specification, especially through the port name which denotes the blackboard name. The export mechanism for blackboard ports of blops works analogously.

To summarize, the potential metaphor in the model permits to treat ports homogeneously for both blops and processes. A single abstraction, the port, is used to denote various communication structures in which processes and blops can get involved interchangeably.

2.3.5 Static Ports

The creation of static ports and their potentials in the blop is done automatically upon start-up of a process or blop. The blop is responsible for matching ports. Seen from this point of view, a blop performs a certain activity, upon creation of a new process or blop it matches as many static ports as possible.

2.3.6 Dynamic Ports

As already stated, dynamic ports will be created in the computation language; they are therefore created by processes only. Their type must be specified in the coordination language. The creation of a dynamic port results in a new potential in the current blop.

2.4 Connections

Connections between processes have either stream semantics, in form of point-to-point communication, group, or blackboard semantics.

- Point-to-point Stream. `1:1`, `1:n`, `n:1` and `n:m` communication patterns are possible;
- Group. Messages are broadcasted to all members of the group;
- Blackboard. Messages are placed on a blackboard used by several processes.

A possible extension of the model would be that a group or blackboard port is connected to more than one group or blackboard, just like the ports for streams. This however would require a wildcard construct to specify the group or blackboard name in the port. From the process point of view this can be achieved by using more ports for each blackboard or group.

Condition on Ports	Explanation
`accessed(p)`	Port was accessed
`unbound(p)`	No comm. partners
`isempty(p)`	Contains no data
`isfull(p)`	Port is full
`msg_handled(p, int n)`	n msg. handled
`less_msg_handled(p, int n)`	$\leq$ n msg. handled

Table 2: Conditions on ports, p denotes a port.

2.5 Events

An event handler may be attached to a condition which determines when the event will be executed in the blop. The conditions are related to ports of processes or blops. Whether an event must be triggered or not will be checked by the system if and only if data flows through the port 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. The event is handled by an event handler inside the blop.

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. This is usually done in the event handling routine of the event currently processed.

The unbound condition on ports permits to construct parallel software pipelines very elegantly. By attaching an appropriate event handler to an initially unbound port of a process, a new process can get created automatically. The mechanism can then be recursively applied to the new process.

2.6 Primitives

STL is a separate language used in addition to a given computation language, however the coordination mechanisms must be accessed from within the computation language. This is done by providing a set of primitives which enable the interaction between the computation and coordination parts of the distributed application. We use `port_export()` to dynamically create new ports from within a process and a set of communication functions to send and receive data via the port. The semantics of communication primitives is dependent on the port type. For blackboard ports, messages are named and tagged on the blackboard and can be read and/or removed from it (using `get()`, `put()`,

`read()` and predicates `readp()`, `getp()` with the usual semantics) whereas for stream and group ports messages are only tagged. A message itself is a compound data structure consisting of several basic data types (`int`, `float`, `double` and the like); sender and receiver must use the same message format for message exchange. Dynamic process management within the computation language is supported by a `create_process()` procedure to which a valid process type must be supplied. For details see [16].

3 Coordination of Autonomous Agents in STL

Works in *Autonomous Agents* constitute a whole discipline of *Artificial Intelligence*, whose description would be prohibitive to do here; as it is not the main concern of this paper, only the concepts of *Autonomous Agents* necessary to understand our implementation will be presented. More information can be found in [17] and [22]. We will focus exclusively on autonomous agents that are considered to be embodied systems, which are designed to fulfill internal or external goals by their own actions in continuous long-term interaction with the environment (possibly unpredictable and dynamical) in which they are situated.

3.1 A Generic Model for an Autonomous Agents' System

Our generic 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 *Neighbor Cells*, which implicitly sets the topology and a list of on-cell available *Objects* a at a given time. This way of encoding the environment allows the user to cope with any type of topology, be it regular or not, since for every cell the number of neighbors can be specified. Note that a cell can contain a region made up of a set of continuous points, e.g. for simulating an area with real coordinates rather than discrete ones.

The architecture of an agent is displayed on Figure 2. 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 2, namely *Perception*, *State*, *Actions* and *Control Algorithm* depends on the application and is the user's responsibility. The *Control Algorithm* module is particularly important because it defines the type of autonomy [22] of the agent: 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.

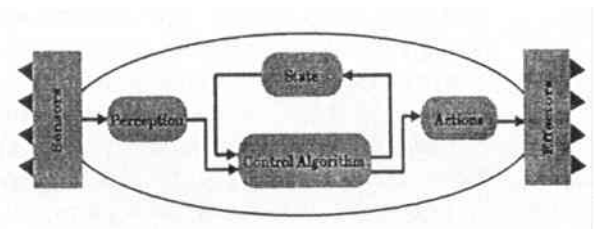

Figure 2: Architecture of an agent.

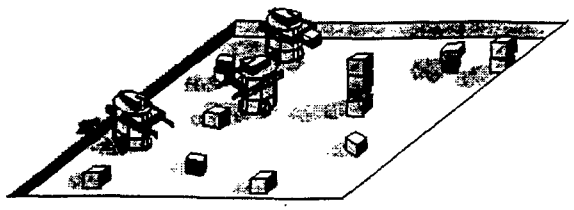

Figure 3: Collective robotics application: stacking objects.

3.2 A Typical Application

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 3. The innovative aspect of our approach rests on a system integrating autonomous agents, that is, every agent in the system has the freedom to act on a cell (the agent decides by itself which action to take). 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 [10]. An implementation in a real world using real mobile robots is currently being developed.

3.3 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. As the *Environment* and the list of *Agents* will be distributed, we will need to develop two types of mechanisms: some in

order to cope with agents crossing borders between sub-environments (of course this should be achieved transparently to the user, it should be part of the software platform), and others in order to cope with data consistency (e.g. updating the number of objects on a cell). We will need some flexible coordination tools that will not alter every agent's autonomy and behavior: we will have to dismiss any unnecessary dependency.

3.4 Preliminary Implementation in STL

The *Environment* is a torus grid with a four connectivity (each cell has four neighbors). Agents comply rigorously with the model previously introduced (Figure 2). 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 4, 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.

For our implementation we introduce four variants of the P2P port type, described hereafter. `P2P<-*` and `P2P->*` are identical to `P2P<-` and `P2P->` except that the `saturation` attribute is set to 2. `P2P1<-N` is an input P2P port capable of matching with `N` other `P2P->` or `P2P->*` ports. `P2P1->N` is an output P2P port capable of matching with `N` other `P2P<-` or `P2P<-*` ports.

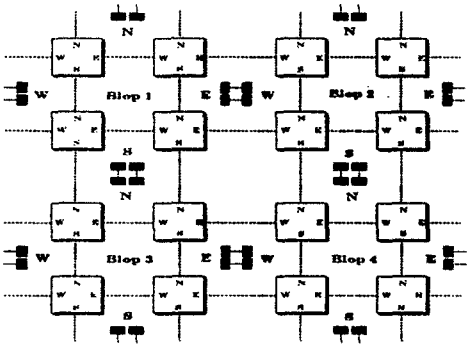

Figure 4: Splitting an environment made up of cells into four blops.

3.4.1 Global Structure

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

The *init* process has two static ports (of type `P2P->`) for every blop to be initialized (Figure 5 illustrates the connections between the *init* process and a blop *se*). The rôle of the *init* process is twofold: first, to create through its *cre_Agts* port the initial agents within every blop; secondly, to set up through its *cre_SubEnv* port the sub-environment (size, number of objects, etc.) of every blop.

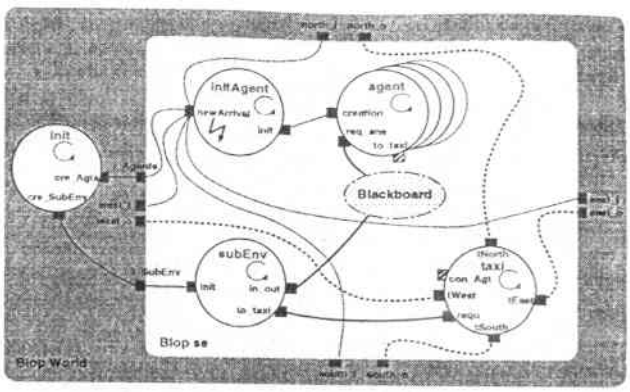

Figure 5: *init* process and blop *se*: solid and dotted lines are introduced just for a purpose of visualization.

Blop *se*: Figure 5 shows the basic organization of processes within a blop *se* and their coordination through ports. Figure 6 diplays the implementation of *se* in STL. Two types of processes may be distinguished: processes that are part of the multi-agent coordination platform, namely *initAgent* and *taxi*, and processes that are intrinsic to the application, viz. *subEnv* and *agent* processes.

Ports of a Blop: Each blop has ten static ports: four `P2P->*` ***outflowing direction*** ports (*north_o, south_o, west_o, east_o*) and four `P2P<-*` ***inflowing direction*** ports (*north_i, south_i, west_i, east_i*), which are used for agent migration, and two `P2P<-*` ports, namely *i_Agents* and *i_SubEnv* used respectively for the creation of the initial agents and for the initialization of the *subEnv* process.

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

initAgent* Process, *newAgentEvt* Event:** The *initAgent* process (C++ code in Figure 7) is responsible for the creation. It has two static ports: *newArrival* and *init*. The *newArrival* `P2P1<-N` port is connected to all ***inflowing direction ports of the blop within which it resides. As soon as a value comes to this port, the *initAgent* process copies it onto its *init* `P2P1->N` port. In the meantime, the *newAgentEvt* event (see Figure 6) is triggered and it will create a new *agent* process, which through its *creation* port

```
blop se(PORTS north_o north_i south_o south_i
         west_o west_i east_o east_i
         i_SubEnv i_Agents,
       VALUES name n s w e) {
  P2P->* north_o(n);   P2P<-* north_i(n);
  P2P->* south_o(s);   P2P<-* south_i(s);
  P2P->* west_o(w);    P2P<-* west_i(w);
  P2P->* east_o(e);    P2P<-* east_i(e);
  P2P<-* i_SubEnv("INIT-SE-" + name);
  P2P<-* i_Agents("INIT-A-" + name);
  process initAgent(PORTS newArrival init) {
    P2P1<-N newArrival("INIT-A-" + name,
                       n, s, w, e);
    P2P1->N init("AGENT-INIT");
  }
  process agent(PORTS creation req_ans) {
    P2P<- creation("AGENT-INIT");
    BB req_ans("SUBENV-AGENT");
  }
  process subEnv(PORTS init in_out to_taxi) {
    P2P<- init("INIT-SE-" + name);
    BB in_out("SUBENV-AGENT");
    P2P-> to_taxi("TAXI");
  }
  process taxi(PORTS tNorth tSouth tWest tEast
                                        requ) {
    P2P-> tNorth(n);  P2P-> tSouth(s);
    P2P-> tWest(w);   P2P-> tEast(e);
    P2P<- requ("TAXI");
  }
  event newAgentEvt {
    create process agent a;
  }
  create process subEnv env;
  create process taxi tx;
  create process initAgent i;
  when accessed(i.newArrival) then newAgentEvt;
}
```

Figure 6: Implementation of the blop *se* in STL.

will read the value that was previously written on the *init* port of the *initAgent* process. Values that are transmitted feature for instance the *state* of the agent to create.

***agent* Process:** This process (C++ code in Figure 8) has two static ports (*req_ans* of type BB and *creation* of type P2P<-) plus *to_taxi* a dynamic P2P-> port. As already stated, this process reads on its *creation* port some values (its *state*). All *req_ans* ports of the agents are connected to a *Blackboard*, through which agents will sense their environment (*perception*) and act into it (*action*), by performing *put/get* operations (Linda-like *in/out*) with appropriate messages. The type of action depends on the type of *control Algorithm* implemented within the agent (see the architecture of an agent on Figure 2). The *to_taxi* port is used to communicate dynamically with the *taxi* process in case of migration: the *state* of the agent is indeed copied to the *taxi* process. The decision of migrating is always taken by the *subEnv* process.

```
void initAgent(P2P1<-N newArrival, P2P1->N init) {
  ByteTempl<32> state;
  Msg stateTp(state);
  while (TRUE) {
    newArrival.get(0, stateTp);
    init.put(0, stateTp);
  }}
```

Figure 7: Implementation of *initAgent* in C++.

```
void agent(BB req_ans, P2P<- creation) {
  ByteTempl<32> state, answer;
  ByteObject<32> *req;
  Msg stateTp(state);           // Message
  boolean noMigration = TRUE;
  creation.get(0, stateTp);     // Initialize
  while (noMigration) {
    req = make_req();           // Perception/Action
    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);
  }
  P2P-> to_taxi;                // For migration
  to_taxi.port_export("MIG" + req->id);
  to_taxi.put(0, stateTp);      // Transfer state
  exit(0);                      //  to taxi
}
```

Figure 8: Implementation of *agent* in C++.

***subEnv* Process:** The *subEnv* process (C++ code in Figure 9) 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 P2P-> port *to_taxi* connected to the *taxi process*. Once initialized through its *init* P2P<- port, the *subEnv* process builds the sub-environment. By performing *put/get* 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 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).

The *taxi* Process: The *taxi* process (C++ code in Figure 10) is responsible for migrating agents across blops. It has four static *direction* ports (of type P2P->), which are connected to the four *outflowing direction* ports of the blop within which it stands. When this process receives on its static P2P<- port

```
void subEnv(P2P<- init, BB in_out, P2P-> to_taxi) {
  IntTempl id, nbOfAgt, myNbOfObj, nbOfCell, pos;
  ByteTempl<32> req, *resp;
  SubEnv *subenv;
  Msg initTp(nbOfAgt, myNbOfObj, nbOfCell, pos);
  init.get(0, initTp);
  // Build the sub-environment
  subenv = init_env(nbOfAgt, nbOfObj, nbOfCell);
  while(TRUE) {                  // Request-Answer
    Msg requestTp("request", id, req);
    in_out.get(0, requestTp); // Get request
    resp = decide_response(req, subenv);
    Msg answerTp("answer", id, *resp);
    in_out.put(0, answerTp);  // Put answer
    if migrateP(resp) {       // Agent migrates
      Msg migTp(id, CharObject<4>(getDir(req)));
      to_taxi.put(0, migTp);  // Inform taxi
}}}
```

Figure 9: Implementation of *subEnv* in C++.

```
void taxi(P2P-> tNorth, P2P-> tSouth, P2P-> tWest,
          P2P-> tEast, P2P<- requ) {
  CharTempl<4> direction;
  IntTempl id;
  ByteTempl<32> state;
  Msg stateTp(state);
  Msg init(id, direction);
  while(TRUE){
    requ.get(0, init);        // Init. from subEnv
    P2P<- con_Agt;
    con_Agt.port_export("MIG" + id);
    con_Agt.get(0, stateTp); // Get agent's state
    switch (direction) {      // Migration
      case "N":               // Handle directions
        tNorth.put(0, stateTp);
        break;
      ...
}}}
```

Figure 10: Implementation of *taxi* in C++.

requ the direction towards where this agent has to migrate, it will create a dynamic P2P<- port *con_Agt* in order to establish with the appropriate *agent* process a communication, by means of which it will collect all the useful information of the agent (*state*). These values will then be written on the port corresponding to the direction to take and will be transferred to 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.

4 Conclusion

In this paper, we presented STL our coordination model and corresponding language. Although the coordination model has some similarities with MANIFOLD, ConCoord, Darwin or Linda (in-depth comparisons can be found in [21]), it however differs in several points. Firstly STL allows the user to define several different port types, yielding to different communication metaphors like generative communication or point to point message passing. Secondly, by using a nested description language to specify different hierarchies of coordination spaces, STL's hierarchical coordination model seems to be more explicit than the one used in MANIFOLD for example. Blops not only serve as a sort of coordinator process which controls coordination, but also as a separate name space for port objects and modularization mechanism for event handling.

We built a coordination platform based on STL's coordination model. This first prototype has been developed on top of the existing PT-PVM platform [15]. A preliminary 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 subtyping of 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 coordi-

nation 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 the reviewers who, thanks to their comments, significantly improved the quality of the paper.

References

[1] G. Agha, S. Folund WooYoung, and Kim Rajendra Panwar. Abstraction and Modularity Mechanisms for Concurrent Computing. *IEEE Parallel & Distributed Technology*, 1(2):3–14, May 1993.

[2] F. Arbab, I. Herman, and P. Spilling. An Overview of Manifold and its Implementation. *Concurrency: Practice and Experience*, 5(1):23–70, February 1993.

[3] Farhad 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.

[4] J.P. Banâtre and D. Le Métayer. Programming by Multiset Transformation. *Communications of the ACM*, 36(1):98–111, 1993.

[5] 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.

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

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

[8] 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.

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

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

[11] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

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

[13] 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.

[14] T. Kielmann. Designing a Coordination Model for Open Systems. In Paolo Ciancarini and Chris Hankin, editors, *First International Conference on Coordination Models, Languages and Applications*, number 1061 in LNCS. Springer Verlag, April 1996.

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

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

[17] P. Maes. Behavior-Based Artificial Intelligence. In *Proceedings of the Fifteenth Annual Meeting of the Cognitive Science Society*, pages 74–83, Hillsdale, NJ, 1993. Lawrence Erlbaum.

[18] Jeff Magee, Naranker Dulay, and Jeff Kramer. Structuring parallel and distributed programs. *Software Engineering Journal*, pages 73–82, March 1993.

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

[20] 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.

[21] M. Schumacher, F. Chantemargue, T. Dagaeff, 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.

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