Autonomous Agents: from Concepts to Implementation*

F. Chantemargue, O. Krone, M. Schumacher, T. Dagaeff, B. Hirsbrunner
Computer Science Department, PAI group, University of Fribourg
Pérolles 3, CH-1700 Fribourg, Switzerland
http://www-iiuf.unifr.ch/pai

Abstract

A model for autonomy-based multi-agent systems aimed at exhibiting emerging properties is proposed. Then, the prerequisites for a distributed implementation are discussed. A preliminary distributed implementation, illustrated by an application to a robotics simulation, is consequently sketched with a strong emphasis on STL, our coordination model, whose aim is to provide powerful coordination mechanisms that do not alter the model's conceptual prescriptions.

1 Introduction

Artificial Intelligence (*AI*) aims at synthesizing intelligence in artefacts. However two families of approaches exist disagreeing in their notion of what intelligence actually means [Ziemke, 1997], [Franklin, 1995]. On the one hand, *Top-Down AI* considers intelligence as the capacity to form and manipulate internal representational models of the world. On the other hand, *Bottom-Up AI* (or *Autonomous Agents*) considers intelligence as a biological feature [Maturana and Varela, 1980]; this notion is often referred to as *Enactivism*.

There is a vast number of papers dealing with *Autonomous Agents*. Our aim is not to go through all of them in depth, but rather to briefly introduce the necessary notions with which our work is related. Autonomous agents are by definition considered to be embodied systems (for the different forms of embodiment, see for instance [Brooks, 1991], [P. Lerena and M. Courant, 1996], [Robert *et al.*,] and [Nwana, 1996]). They 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. Dealing with interactions leads naturally to the concept of emergence of behavior and/or functionality. Emergence offers indeed a bridge between the necessity of complex and adaptive behavior at a macro level and the mechanisms of multiple competences and situation-based learning at a micro level. A system's behavior can be considered emergent if it can only be specified using descriptive categories which are not to be used to describe the behavior of the constituent components.

2 Our Multi-Agent Model

Our model for autonomy-based multi-agent systems is composed of an *Environment* and a list of *Agents*. The *Environment* encompasses a list of *Cells* and a set of *Objects* which will be manipulated by the agents. Every *Cell* contains a list of *Neighbour Cells*, which implicitly sets the topology, and the set of objects actually available on it at a given time.

The architecture of an agent is displayed on figure 1. An agent possesses some sensors to perceive the world within which it moves, and some effectors to act in this world, so that it complies with the prescriptions of *physically embodied agents* and *simulated embodied agents* [Ziemke, 1997]. The implementation of the different modules presented on Figure 1, namely *Perception*, *State*, *Actions* and *Control Algorithm* depends on the application and is the user's responsibility. In the *Perception* module, the designer specifies the type of perception of the agent, e.g. if the agent perceives only the number of objects on the cell on which it stands. The *State* module encompasses the private information of the agent, e.g. whether it carries or not an object, its strain or whatever. The *Actions* module typically consists of the basic actions the agent can take, e.g. move to next cell, pick up an object or drop an object. The *Control Algorithm* module is particularly important because it defines the type of autonomy of the agent: it is precisely inside this module that the designer decides whether to implement an operational autonomy or a behavioral autonomy [Ziemke, 1997]. Operational autonomy is defined as the capacity to operate without human intervention, without being

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

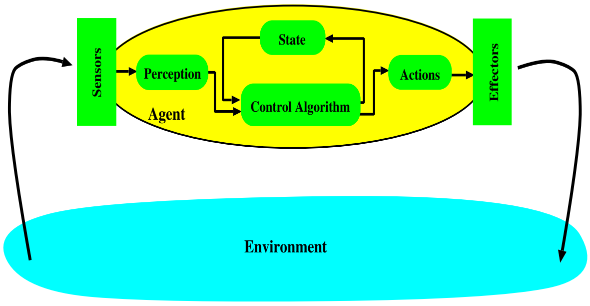

Figure 1: Architecture of an agent.

remotely controlled. Behavioral autonomy supposes that the basis of self-steering originates in the agent's own capacity to form and adapt its principles of behavior: an agent, to be behaviorally autonomous, does not only need the freedom to behave/operate without human intervention (operational autonomy), but further the freedom to have formed (learned or decided) its principles of behavior on its own (from its experience), at least partly.

3 Distributed Implementation

3.1 The Model's Prerequisites

Our multi-agent model is aimed at addressing problems which are naturally distributed (both in space and time). In our model, agents coexist (in space and time) and share a common environment. Moreover, agents can move, act in parallel in the environment, each one having its own freedom for deciding. This implies a certain independence between agents in the system and between areas of the environment. Moreover, it is assumed that each agent has its own time or temporality, which is peculiar to its identity. To express our autonomy-based multi-agent model on a distributed architecture in the most natural way, we need an appropriate software platform in which the implementation would be the image of the model (the concepts) with the highest fidelity. We need some tools allowing us to coordinate our autonomous agents (since they interact in their environment) while preserving their structure and their behavior, like for instance their degree of autonomy.

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

We have therefore to identify some flexible coordination patterns that will not alter every agent's temporality and behavior, and we have to develop corresponding tools and primitives. Our platform must be able to render to some extent the independence in space and time, without being interfered by coordination mechanisms exclusively embedded for a purpose of implementation.

3.2 Coordination Models

Overview

Today's state of the art parallel programming models used for implementing general purpose distributed applications suffer from limitations concerning a clean separation of the computational part and the "glue" that coordinates the overall distributed application. Especially these limitations make a distributed implementation of autonomy-based multi-agent models, our concern, a burdensome task. To study problems related to coordination, Malone [Malone and Crowston, 1994] introduced a new theory called *coordination theory* aimed at defining such a "glue". The key issue of coordination is *managing dependencies among activities.* To formalize and better describe these interdependencies it is necessary to separate the *computation* and the *coordination* of a parallel application [N. Carriero and D. Gelernter, 1992]. The research in this area has led to the definition of several coordination models and corresponding coordination languages, whose most prominent representative is Linda [Gelernter, 1985]. Other models and languages are based on message passing paradigms [Agha *et al.*, 1993], object-oriented techniques [Kielmann, 1996], multi-set rewriting schemes [Banâtre and Métayer, 1993] or control-driven models [Arbab *et al.*, 1993].

Our Coordination Model: STL

The coordination model of STL (Figure 2) shares few characteristics with the IWIM [Arbab, 1996] model of coordination. It comprehends five building blocks (details in [Krone *et al.*, 1998]):

- *Blops*, as an abstraction and modularization mechanism for processes and ports. It serves as a separate name space as well as an encapsulation mechanism for events. Blops have the same interface as processes, i.e., a name and a set of ports. The creation of blops is handled in the same way as the creation of processes. It includes the initialization of all processes and ports defined for this blop and subordinated blops.

- *Processes*, as a representation of active entities. A process is a typed object, it has a name and a set of ports. Processes in STL do not know any kind of process identification, instead a black box process model is used. Process termination is implicit.

- *Ports*, as the interface of processes/blops to the external world. Every communication in STL is handled via a connection which is the result of ports matching. A port has a *name* and a set of well defined *attributes*: they are referred to as the port's *signature*. The combination of port attributes results in a port type. Examples of port attributes are `saturation` or `communication`. The former attribute defines the number of ports that may connect to it. The latter attribute relates to the communication paradigms that can be chosen: `blackboard`, `group` or `stream`. Several port types exist: the `BB` type (which has a tuple space semantics à la Linda), the `Group` type and `P2P` type (point to point) [Krone, 1997]. Variants of these types can be defined by the user. Thus, we introduced several variants of the `P2P` port type: `P2P_i` and `P2P_o` ports are respectively input and output `P2P` ports with `saturation` set to 1; `P2P_iN` and `P2P_oN` are respectively input and output `P2P` ports with `saturation` set to `N`. To match, pairs of ports must have compatible signatures, thus introducing a sub-typing relation on port signatures [Krone *et al.*, 1998].

- *Events*, as a mechanism to dynamically react to state changes within a blop. They can be triggered using a condition operation on a port. STL provides some conditions such as `unbound`, `isfull` or `accessed`.

- *Connections*, as a representation of connected ports. Their semantics depend on the port types.

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, implemented as threads;

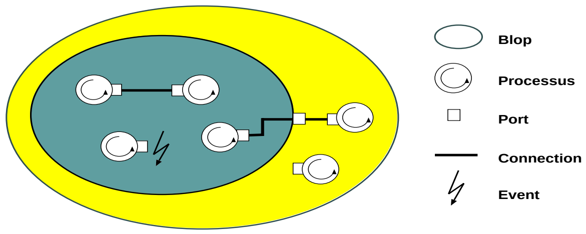

Figure 2: The Coordination Model of STL.

- The *Coordination Media* of STL are the following: events, ports and connections serve as the medium which enables coordination, and a blop is the medium in which coordination takes place;

- The *Coordination Laws* are defined through the semantics of the *Coordination Tools* and the semantics of the interactions with the coordination media by means of events.

STL 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. As far as the implementation of these primitives is concerned, we use PT-PVM [Krone *et al.*, 1996] a software platform for programming multi-threaded applications on a cluster of workstations as the underlying communication and process management platform.

3.3 Application of STL

The Framework

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

We implemented a serial version of this simulation in the Swarm Simulation System [Langton *et al.*, 1996], stressing on the measurement of quantitative results and on the realization of appropriate visualization tools to follow in real-time the run of an experiment. We implemented several variants for agent modules and realized an intensive number of experiments whereof we observed an implicit cooperation between the agents in the system to accomplish a global task, i.e. regrouping objects. In such an experiment, the location of the stack containing the objects at the end of the run is the result of the agents' interactions. Details on the implementation and results can be found in [F. Chantemargue *et al.*, 1996] and [Dagaeff *et al.*, 1997]. A preliminary implementation of this simulation in a real world involving real mobile robots (kheperas) exhibited the emergence of cooperation.

A Preliminary STL-based Implementation

For this preliminary implementation, the *Environment* is made up of a torus grid with a four connectivity (each cell has four neighbors). Agents comply rigorously with the model previously introduced (Figure 1). 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, thus providing an independent functioning between sub-environments. Note that blops have to be arranged in accordance with the topology of the environment they implement.

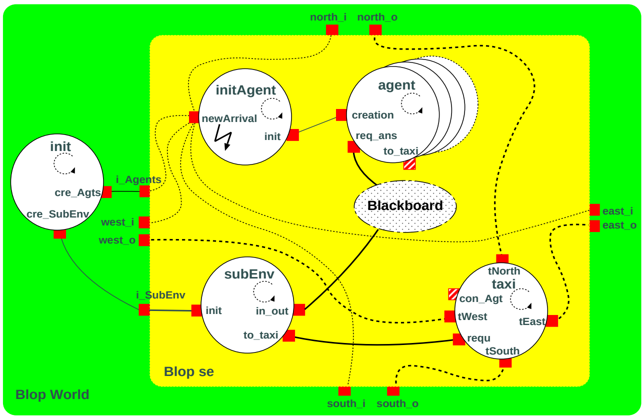

Figure 3: *init* process and blop *se*: solid and dotted lines are introduced just for a purpose of visualization

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

The *init* process has two static ports (of type P2P_o) for every blop to be initialized (Figure 3). 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.

Blop *se*. Figure 3 shows the basic organization of processes within a blop *se* and their coordination through ports. Figure 4 displays the implementation of *se* in STL. Two types of processes may be distinguished: processes that are part of the coordination platform (implementation purpose), namely *initAgent* and *taxi*, and processes that are intrinsic to the multi-agent model, namely *subEnv* and *agent* processes.

```
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_o north_o(n);   P2P_i north_i(n);
  P2P_o south_o(s);   P2P_i south_i(s);
  P2P_o west_o(w);    P2P_i west_i(w);
  P2P_o east_o(e);    P2P_i east_i(e);
  P2P_i i_SubEnv("INIT-SE-" + name);
  P2P_i i_Agents("INIT-A-" + name);
  process initAgent(PORTS newArrival init) {
    P2P_iN newArrival("INIT-A-" + name,
                                   n, s, w, e);
    P2P_oN init("AGENT-INIT");
  }
  process agent(PORTS req_ans creation) {
    P2P_i creation("AGENT-INIT");
    BB req_ans("SUBENV-AGENT");
  }
  process subEnv(PORTS init in_out to_taxi) {
    P2P_i init("INIT-SE-" + name);
    BB in_out("SUBENV-AGENT");
    P2P_o to_taxi("TAXI");
  }
  process taxi(PORTS tNorth tSouth tWest tEast
                                        requ) {
    P2P_o tNorth(n);  P2P_o tSouth(s);
    P2P_o tWest(w);   P2P_o tEast(e);
    P2P_i requ("TAXI");
  }
  event newAgentEvt {
    create process agent a;
    when accessed(i.newArrival) then newAgentEvt;
  }
  create process subEnv env;
  create process taxi tx;
  create process initAgent i;
  when accessed(i.newArrival) then newAgentEvt;
}
```

Figure 4: Implementation of the blop *se* in STL.

Ports of a Blop. Each blop has ten static ports: four P2P_o *out-flowing direction* ports (*north_o, south_o, west_o, east_o*) and four P2P_i *in-flowing direction* ports (*north_i, south_i, west_i, east_i*), which are used for agent migration, and two P2P_i 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 *in-flowing direction* ports of a blop match with ports of its inner process *initAgent*. The four *out-flowing direction* ports of a blop match with ports of its inner process *taxi*.

initAgent **Process,** *newAgentEvt* **Event**. The *initAgent* process is responsible for the creation of agents. It has two static ports: *newArrival* and *init*. The *newArrival* P2P_iN port is connected to all *in-flowing direction* ports of the blop within which it resides. As soon as a value comes to this port, the *initAgent* process copies it onto its *init* P2P_oN port. In the meantime, the *newAgentEvt* event (see Figure

```
void agent(BB req_ans, P2P_i creation) {
  ByteTempl<32> state, answer;
  ByteObject<32> *req;
  Msg stateTp(state);      // Message
  boolean noMigration = TRUE;
  creation.get(0, stateTp);// Initialize
  while (noMigration) {    // Perception/Action
    req = make_req();
    Msg requestTp("request", req->id, *req);
    req_ans.put(0, requestTp); // Put request
    Msg answerTp("answer", req->id, answer);
    req_ans.get(0, answerTp);  // Get answer
    control(answer);       // Control Algorithm
    state = update_state(answer);
    noMigration = migrate_p(answer);
  }
  P2P_o to_taxi;           // For migration
  to_taxi.port_export("MIG" + req->id);
  to_taxi.put(0, stateTp); // Transfer state
  exit(0);                 // To taxi
}
```

Figure 5: Implementation of *agent* in C++.

4) is triggered and it will create a new *agent* process, which through its *creation* port will read the value that was previously written on the *init* port of the *initAgent* process. Transmitted values are for instance the *state* of the agent to create.

agent **Process**. This process (C++ code in Figure 5) has two static ports (*req_ans* of type `BB` and *creation* of type `P2P_i`) plus *to_taxi* a dynamic `P2P_o` 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 *out/in*) with appropriate messages. The type of action depends on the type of *control Algorithm* implemented within the agent (see Figure 1). The *to_taxi* port is used to communicate dynamically with the *taxi* process in case of migration: the *state* of the agent is indeed copied to the *taxi* process. The decision of migrating is always taken by the *subEnv* process.

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 `P2P_o` port *to_taxi* connected to the *taxi process.* Once initialized through its *init* `P2P_i` 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 is responsible for migrating agents across blops' boundaries. It has four static *direction* ports (of type `P2P_o`), which are connected to the four *out-flowing direction* ports of the blop within which it stands. When this process receives on its static `P2P_i` port *requ* the direction towards where the agent has to migrate, it will create a dynamic `P2P_i` 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 and Future Works

We presented a model for autonomy-based multi-agent systems and its prerequisites for parallelization. We built a coordination platform based on STL's coordination model on top of the existing PT-PVM platform [Krone *et al.*, 1996]. We sketched a preliminary STL-based distributed implementation of our multi-agent model applied to a collective robotics simulation. STL showed its power and demonstrated its appropriateness for coordinating a generic class of autonomous agents, whose most critical constraint is the preservation of temporality by dismissing coordination mechanisms exclusively embedded for purpose of implementation.

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, making this work a step forward in the *Autonomous Agents* community. Secondly, as the generic patterns of coordination for autonomy-based multi-agent implementations are embedded within the platform, a user can quite easily develop new applications (e.g. by changing the type of autonomy of the agents, the type of environment), insofar they comply with the generic model.

Future works are as follows. First, a graphical user interface will be developed in order to facilitate the specification of the coordination part of a distributed application. Secondly, the basic mechanisms of STL have to be enhanced in order to simplify the expression of the coordination and to establish a well defined semantics. Thirdly, the implementation of STL is still to be carried on in two ways: as a separate coordination language and as a coordination library. Fourthly, the STL coordination model is still to be extended in order to encompass as many as possible generic patterns of coordination, yielding in STL templates at disposal for general purpose implementations. In our study, we focused only on operational autonomy in an engineering approach. Future works will consist in studying behavioral autonomy through learning approaches.

References

[Agha *et al.*, 1993] 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.

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

[Arbab, 1996] Farhad Arbab. The IWIM Model for Coordination of Concurrent Activities. In Paolo Ciancarini and Chris Hankin, editors, *Proceedings of the First International Conference on Coordination Models, Languages and Applications*, number 1061 in LNCS. Springer Verlag, April 1996.

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

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

[Dagaeff *et al.*, 1997] 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.

[F. Chantemargue *et al.*, 1996] F. Chantemargue, T. Dagaeff, M. Schumacher, and B. Hirsbrunner. Coopération implicite et performance. In *Proceedings of the Sixth symposium on Cognitive Sciences (ARC)*, Villeneuve d'Ascq, France, December 10–12 1996.

[Franklin, 1995] S. Franklin. *Artificial Minds*. Bradford Books/MIT Press, Cambridge, MA, 1995.

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

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

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

[Krone *et al.*, 1998] O. Krone, F. Chantemargue, T. Dagaeff, M. Schumacher, and B. Hirsbrunner. Coordinating Autonomous Entities. In *Proceedings of the ACM Symposium on Applied Computing (SAC'98). Special Track on Coordination, Languages and Applications*, Atlanta, Georgia, USA, February 27 - March 1 1998. To appear.

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

[Langton *et al.*, 1996] C. Langton, N. Minar, and R. Burkhart. The Swarm simulation System: a toolkit for building Multi-agent simulations. Technical report, Santa Fe Institute, 1996.

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

[Maturana and Varela, 1980] H. Maturana and F.J. Varela. *Autopoiesis and Cognition: the realization of the living*. Reidel, Boston, MA, 1980.

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

[Nwana, 1996] H. S. Nwana. Software Agents: an Overview. *Knowledge Engineering Review*, 11(3):205–244, 1996.

[P. Lerena and M. Courant, 1996] P. Lerena and M. Courant. Bio–machines. In *Proceedings of Artificial Life*, volume V, Nara, Japan, 1996.

[Robert *et al.*,] A. Robert, F. Chantemargue, and M. Courant. Emuds: Virtual worlds for artificial agents. Submitted to ECAI'98.

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