THE STL++ COORDINATION LANGUAGE:
APPLICATION TO SIMULATING THE AUTOMATION OF A TRADING SYSTEM

Michael Schumacher, Fabrice Chantemargue, Simon Schubiger, Béat Hirsbrunner
University of Fribourg, Computer Science Department, PAI group
Pérolles 3, CH-1700 Fribourg, Switzerland
Email:{FirstName.LastName}@unifr.ch, URL: http://www-iiuf.unifr.ch/pai

Oliver Krone
Swisscom, Information Technology and Applications
CH-3000 Bern, Switzerland
Email:Oliver.Krone@swisscom.com

Key words: Coordination, Multi-Agent Systems, Agent-Oriented Programming, Distributed Artificial Intelligence Applications, Communication Models.

Abstract: This paper introduces the STL++ coordination language, a C++-based language binding of the ECM coordination model. STL++ applies theories and techniques known from coordination theory and languages in distributed computing to try to better formalise communication and coordination in distributed multi-agent applications. STL++, as such, may be seen as a preliminary agent language which allows the organisational structure or architecture of a multi-agent system to be described, with means to dynamically reconfigure it. It is aimed at giving basic constructs for distributed implementations of generic multi-agent platforms, to be run on a LAN of general-purpose workstations. We illustrate the application of STL++ to a real case study, namely the application to simulating the automation of a trading system.

1. INTRODUCTION

Coordination theory introduced by Malone (Malone & Crowston 1994) is concerned with the management of dependencies between different activities. Tenets developed in this theory encompass conceptual and methodological aspects that enable a distributed application to have a better expressiveness and to be much more easily implemented, through a clear separation between coordination and computation (Carriero & Gelernter 1992). In computer science, research in coordination theory focused on the definition of multiple *coordination models* and related *languages* (Papadopoulos & Arbab 1998). A coordination language is the linguistic embodiment of a coordination model (Carriero & Gelernter 1992) and has to be orthogonal to a computation language, in the sense that it extends the computation language with additional functionalities which facilitate the implementation of distributed applications.

The most representative of this class of languages is Linda (Carriero & Gelernter 1989), which is based on a *tuple space abstraction* as the underlying coordination model. Multiple extensions and applications have been realised, e.g. Bonita (Rawston & Wood 1997), Piranha (Carriero, Freeman, Gelernter & Kaminsky 1995). Other models and languages are based on *control-oriented approaches*, e.g. IWIM/Manifold (Arbab 1996); *message passing paradigms*, e.g. CoLa (Hirsbrunner, Aguilar & Krone 1994); *object-oriented techniques*, e.g. Objective Linda (Kielmann 1997), JavaSpaces (Sun Microsystems 1998); *multi-set rewriting schemes*, e.g. Bauhaus Linda (Carriero, Gelernter & Zuck 1995); or *Linear Logic*, e.g.

Linear Objects (Bourgois, Andreoli & Pareschi, 1992).

Coordination is likely to play a central role in multi-agent systems (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.

Agent languages should have means to clearly describe the coordination part of a MAS application. The coordination language STL++, presented in this paper, applies theories and techniques known from coordination theory and languages in distributed computing to better formalise communication and coordination in distributed multi-agent applications. STL++, as such, may be seen as a preliminary agent language which allows the organisational structure or architecture of a MAS to be described, with some means to dynamically reconfigure it. It is aimed at giving basic constructs to implement more elaborated agent languages with powerful tools.

The numerous activities that take place within a trading system are typically distributed and can be modelised by a multi-agent system. It has led several works to propose solutions for agent-based electronic commerce (Guttman, Moukas & Maes 1998); see for instance Kasbah (Chavez & Maes 1996), Market Maker (Wang 1999), SICS MarketSpace (Eriksson, Finne & Sverker 1998), FishMarket (Rodriguez-Aguilar, Noriega, Sierra & Padget 1997), ZEUS (Collis & Lee 1998), or a proposal based on the PageSpace platform (Ciancarini, Knoche, Tolksdorf & Vitali 1996). This paper presents a simulation of the automation of a trading system and shows how it can be implemented in STL++.

This paper is organised as follows. Section 2 describes the main features of STL++, an instantiation of the ECM model in an object oriented language, namely C++. Section 3 presents the simulation of the automation of a trading system. In the last section some conclusions are drawn and future work is outlined.

2. THE COORDINATION LANGUAGE STL++

STL++ is based on the ECM (Encapsulation Coordination Model) coordination model (Krone, Chantemargue, Dagaeff & Schumacher 1998). It uses an encapsulation mechanism as its primary abstraction (referred to as blops), offering structured separate name spaces which can be hierarchically organised.

An application written using the STL++ library runs on a network of UNIX workstations and consists of a hierarchy of blops in which several agents run. Agents, which are embedded in lightweight processes (threads), communicate anonymously within and/or across blops through connections. The latter are established by the matching of the communication interfaces of these agents.

STL++ consists of five building blocks (see figure 1):

1. *Agents*, the active entities;
2. *Blops*, as an abstraction and modularization mechanism for group of agents and ports;
3. *Ports*, as the interface of agents/blops to the external world;
4. *Events*, a mechanism to support dynamic state changes inside a blop;
5. *Connections*, as a representation of matched ports.

According to the general characteristics of what makes a coordination model and corresponding coordination language (Kielmann & Wirtz 1996), these elements are classified in the following way:

1. The *Coordination Entities* are the agents;
2. 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;

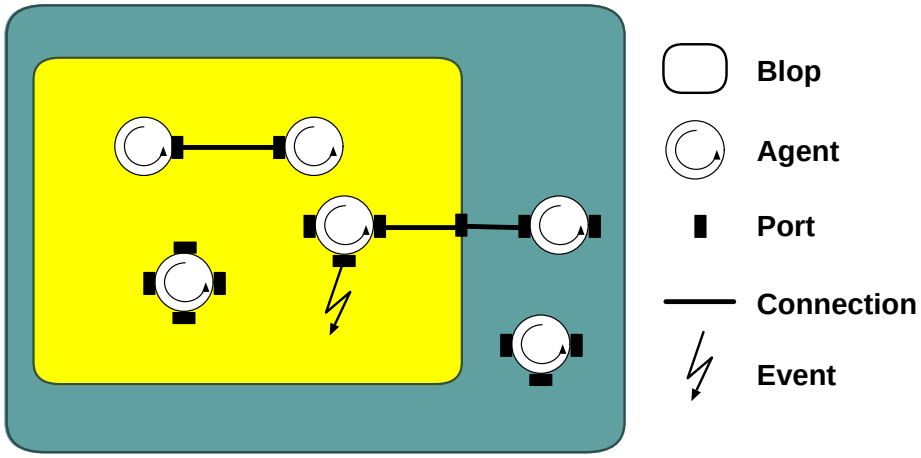

Figure 1. The Key Components of the Coordination Language STL++

3. The *Coordination Laws* are defined through the semantics of the *Coordination Tools* (the operations on the port abstraction) and the semantics of the interactions with the coordination media by means of events.

2.1 Blops

Blops are an abstraction for an agglomeration of agents and blops to be coordinated and serve as a separate name space for ports, agents, and subordinated blops as well as an encapsulation mechanism for events. Blops enable structured hierarchical sets of agent environments to be set up, each of which being a private coordination space with communication interfaces (ports) to other environments. A set of complex interconnected and hierarchically organised spaces is indeed often desirable in MAS.

A blop inherits from the base class `Blop` and has to reimplement the `start()` method. The user must call `Blop::start()` for initialisation. The creation of a blop results in the initialisation of all enclosing blops, ports and agents. New blops can be dynamically created during execution. Blops can be parameterised.

2.2 Agents

An agent has a name and a set of ports (which can be empty). It exclusively communicates *anonymously* through its ports. So, an agent does not have to care about to which agent information will be transmitted or received from. There is no agent identification; instead a black box model is used.

Agent classes inherit from the base class `Agent` and have to reimplement the `start()` method. Agents can be created directly by blops, through events or by other agents. An agent is initialised with `Agent::start()`. To communicate, agents dynamically create ports of predefined types through instantiation of a port class. Agent termination is implicit: the agent disappears along with its ports and data, when the corresponding object terminates.

Realising agents as black boxes is a way to implement autonomous agents. An agent owns exclusive control over its internal state and behaviour; it can define by itself its ports. It is seen from external views (its environment) as a delineated entity presenting clear interfaces. Being distinct from the outside (the environment), an agent is part of its environment composed by its surrounding blop and the agents living in it.

Table 1. Port Features (the first three are primary features, the last three are secondary features)

Feature	Values	Explanation
Communication	`blackboard`, `stream`, `group`	Basic communication paradigms.
Msg. Synchronisation	`synchron`, `asynchron`	Message passing semantics.
Orientation	`in`, `out`, `inout`	Direction of the data flow over a connection.
Saturation	`{1,2,...,INF}`	Number of connections a port can have.
Data Type	`typeName`	Type of data flowing through a port.
Lifetime	`{1,2,...,INF}`	Number of data units that can pass through a port before it decays.

2.3 Ports & Connections

Ports are the interfaces of agents and blops to establish connections to other agents/blops, i.e. communication is handled via a connection and therefore over ports. Ports have one or several *names* and a set of well defined *features* describing the port's characteristics (see Table 1). STL++ distinguishes *Primary* and *Secondary Features*. Primary features define the main semantics of a port. Secondary features refine the port semantics. The combination of port features results in a *port type*. Names and type of a port are referred to as the port's *signature*.

The primitives for accessing data via ports are: `get` (blocking); `put` (nonblocking, except for synchronous ports); and `read` (blocking), only for blackboard ports.

Ports are a manner to implement sensing and acting capacities of agents. In fact, agents should perceive and act through different means, in virtue of the possibility to define port types and several instantiations of them. A port perceives specific data; this is enforced with the data type feature of ports.

Based on the combination of the primary features, four basic port types are defined, resulting in four connection types (see table 2). To ease the implementation, two base port classes for every port type have been defined, one for agents and one for blops.

- **Blackboard ports** (`BB_Port` and `BB_Blop_Port` classes). The resulting connection has a blackboard semantics. The number of participating ports is unlimited. Messages are persistent objects, which can be retrieved using a symbolic name. An agent reading/writing a message does not have to be aware of the agent that has written or that will read this message. Moreover, messages are not ordered.
- **Group ports** (`Group_Port` and `Group_Blop_Port` classes). The resulting connection has a closed group semantics. The number of participating ports is unlimited. Each member of the group can broadcast asynchronously messages to every participant in the group. Messages are stored at the receiver side. Thus, if a port in a group disappears, then the sequence of information that has not been read is lost.
- **S-Stream ports** (`S_inPort`, `S_outPort`, `S_Blop_inPort` and `S_Blop_outPort` classes). The resulting connection has the same semantics as the S-Channel defined in Arbab 1996 (S for synchronous). This connection always results from the matching of uni-directional contradictory oriented ports. In contrast to other connections, this connection never contains data, due to its synchronous nature.
- **KK-Stream ports** (`KK_inPort`, `KK_outPort`, `KK_Blop_inPort` and `KK_Blop_outPort` classes). The resulting connection has an analogous semantics to the asynchronous KK-Channel defined in Arbab 1996 (K for keep), with its specific semantics (see below) when a port disappears from one end of the connection. As for S-Streams, this connection always results from the matching of contradictory oriented ports. If the connection is broken at its consuming port, the next new matching port will consume all pending data. If the connection is broken at its producing port, the consuming port will be able to continue to consume all data in the connection. If both

ports are deleted, the connection disappears with its data.

Table 2. Basic port types with their primary feature's values

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

2.4 Establishing Connections between Agents

Features of pairs of ports must comply with each other for ports to match (see Table 3 and for details see Schumacher, Chantemargue, Krone & Hirsbrunner 1998). Furthermore, four general conditions must be fulfilled for two ports to get matched:

1. both use the same communication paradigm (stream, group or blackboard);
2. both have at least one common 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 (agent or blop).

Conceptually the matching of agent ports can be described as follows. When an agent is created in a blop, it creates with its port signature a "potential" in the blop where it is currently embedded. If two compatible potentials exist in the blop, and if conditions (1)-(4) are fulfilled, the connection between the corresponding ports is established and the potentials disappear.

Table 3. Compatibility for Features `F` for two Ports `P1` and `P2`

Feature `F`	Compatibility for ports `P1` and `p2`
Communication	`P1.F` ≡ `P2.F`
Msg. Synchronisation	`P1.F` ≡ `P2.F`
Orientation	`P1.F` = `in` *and* `P2.F` = `out` *or* `P1.F` = `P2.F` = `inout`
Saturation	Always compatible
Data Type	`P1.F` ≡ `P2.F`
Lifetime	Always compatible

2.5 Events

Events can be attached to conditions on ports (see table 4 for an overview on conditions). A condition determines when the event is triggered.

Event classes inherit from `Event`; the `launch()` method, which defines the acting of the event, must be reimplemented.

Conditions on ports are checked when data flow through the port. Events are instantiated with a specific lifetime, which determines how many times they can be triggered. Agents can also directly raise events.

Table 4. Port Conditions

Condition	Explanation
`UnboundCond()`	Port not connected.
`SaturatedCond()`	Port saturated.
`MsgHandledCond(int n)`	`n` messages handled.
`AccessedCond()`	Port accessed with `put`, `get` or `read` primitives.
`PutAccessedCond()`	Port accessed with `put` primitive.
`GetAccessedCond()`	Port accessed with `get` primitive.
`TerminatedCond()`	Port lifetime is over.

3. SIMULATION OF A TRADING SYSTEM

Although our goal is to fully automate a trading system, for the time being, we rather concentrate on simulating the automation of such a system. Our aim, in this paper, is not to focus on the control algorithms of the different agents, nor on the negotiation techniques (see e.g. Guttman & Maes 1998) that are undertaken by the agents in order to process a transaction, but rather to concentrate on the basic coordination mechanisms that come into play in the interactions between agents, for which STL++ is precisely suitable. Thus, in this implementation, agents are endowed with a very basic autonomy (Ziemke 1997) in the sense that they can make decisions on their own, without user intervention. More sophisticated autonomy-based control algorithms and smart negotiation tech-

niques will be tackled in a further stage, without challenging the work presented in this paper.

Figure 2 gives a scaled down graphical overview of the organisation of the agents that compose our trading system, as well as their interactions. To avoid to overload the graph, port names (on which the matching is based) have been intentionally omitted.

The *TradeWorld* blop confines every activity in the trading simulation. Several *Trading System* blops (TSB) are accessible by *customers* (company or private customers), who are authorised members of a trading system who represent the end-user agents. *Company* or *Private Customers* create queries to buy or sell goods. These queries, written by the customer on his `query_P` port (of `KK_outPort` type), are transmitted to a *Trading System* Blop (TSB).

In a TSB reside *Brokers*, each of which being devoted to serve a particular customer, by

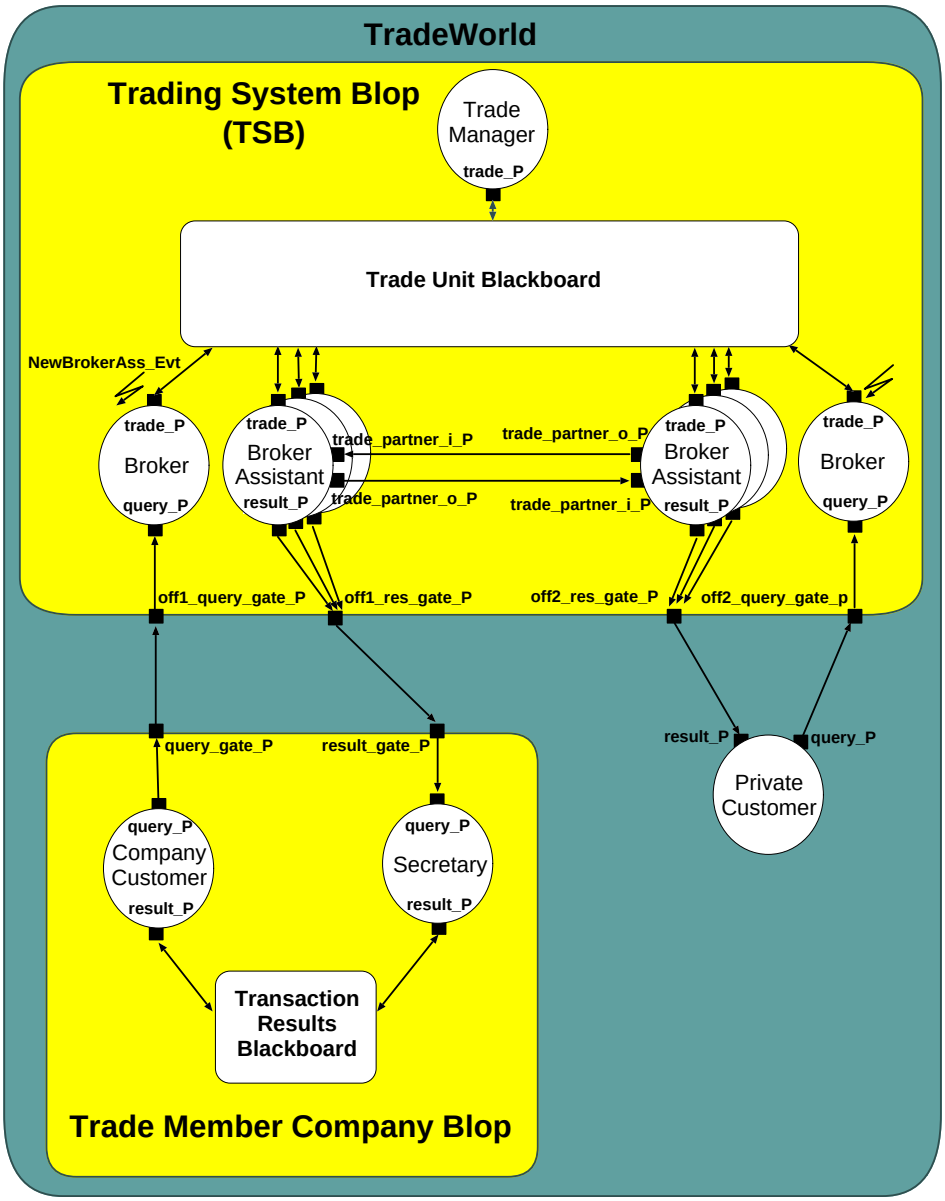

Figure 2. Trading System with STL++

handling his committed customer queries that came in. A pair of ports on the TSB, namely *name*`_query_gate_P` and *name*`_res_gate_P` (*name* being `off1` or `off2`, see figure 2), serves as gates for each customer and his respective broker. Every query is then posted by the broker to his `trade_P` port (of `BB_Port` type), e.g., sell 100 securities at 1000 CHF and therefore published in the *Trade Unit Blackboard*. On the `trade_P` port is bound the event `NewBroker-Ass_Evt` with the condition `PutAccessedCond`; the effect of the posting is that the `NewBroker-Ass_Evt` event is triggered. The role of this event is to dynamically create a *Broker Assistant* that will be in charge of fulfilling the specific query, by establishing a dynamic connection with another broker assistant interested in an (almost) symmetric query (e.g., buy 50 securities at 1000 CHF). This is possible on the basis of the information transmitted by the *Trade Manager*.

All queries are supervised by the *Trade Manager*; he knows who issued which query (through an identification contained in the query). For each new arrived query, the Trade Manager checks whether a possible matching between proposals can take place (e.g. case of a broker who wants to buy securities of type A and another broker who wants to sell securities of type A). Whenever a kind of matching can somehow be issued between two *Broker Assistants*, the Trade Manager puts on the *Trade Unit Blackboard* two appropriate messages for each involved Broker Assistant. The information transmitted contains among others a specific transaction id.

A new created Broker Assistant has first to read on his `trade_P` port (of type `BB_Port`) in order to be informed of a specific transaction. Thanks to this information, he dynamically publishes two `KK_Ports` using the transaction id as port name (`trade_partner_i_P` and `trade_partner_o_P` ports). A new double connection is then established between these two partners. Both involved Broker Assistants exchange useful information so as to make their query successful (this is precisely where negotiation techniques appear).

When a successful transaction (result of the agreement of two Broker Assistants) is issued, both Broker Assistants inform their committed customers by transmitting appropriate information on their `result_P` port (of type `KK_outPort`), and then terminate. On customer side, results about processed queries can be collected either directly or through a *Secretary Agent* (see figure 2). At regular intervals, non-fulfilled queries are eliminated by the Trade Manager; all involved entities are kept posted.

4. CONCLUSION

STL++ is a coordination language, which provides a coordination framework for distributed multi-agent applications. It offers tools to describe the organisational structure or architecture of a MAS, with means to dynamically reconfigure it.

STL++ is still to be extended in order to encompass as much generic coordination patterns as possible, yielding in templates at disposal for general-purpose implementations.

The Trading System simulation should be enhanced so as to tackle more sophisticated autonomy-based control algorithms and smart negotiation techniques.

We are working on a new ECM mapping in Java, using the JavaSpaces (Sun Microsystems 1998) API. This new instantiation will allow applications to be coordinated over the Web, and in particular will be used to implement a full scale Trading System distributed on the Web and accessible through the Web. We plan as well to use this new instantiation to realise the management of a Resource Warehouse (integrated in the Web Operating Project, WOS in short, see Lamine, Plaice & Kropf 1997) with the aim of offering a resource sharing tool over the Web (Schubiger & Krone 1998).

REFERENCES

Arbab, F. 1996, *The IWIM Model for Coordination of Concurrent Activities*. In Ciancarini, P. & Hankin, C.

(eds), Proceedings of the First International Conference on Coordination Models, Languages and Applications, number 1061 in LNCS. Springer Verlag.

Bourgois, M., Andreoli, J.M. & Pareschi, R. 1992, *Extending Objects with Rules, Composition and Concurrency: the LO Experience*. Technical report, European Computer Industry Research Centre, Munich, Germany.

Carriero,N. & Gelernter, D. 1989, *Linda in Context*. Communications of the ACM, 32(4):444-458.

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

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

Carriero, N., Gelernter, D. & Zuck, L. 1995, *Bauhaus Linda*. In Ciancarini, P., Nierstrasz, O. & Yonezawa, A. (eds), Object-Based Models and Languages for Concurrent Systems, volume 924 of Lecture Notes in Computer Science, Berlin. Springer Verlag.

Chavez, A. & Maes, P. 1996, Kasbah: *An Agent Marketplace for Buying and Selling Goods*. In Proceedings of the First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology, London, UK. Lawrence Erlbaum.

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

Collis, J. C. & Lee, L. C. 1998, *Building Electronic Marketplaces with the ZEUS Agent Toolkit*. In Proceedings of 2nd International Conference on Autonomous Agents; Workshop on Agent Mediated Electronic Trading.

Eriksson, J., Finne, N. & Sverker, J. 1998, *SICS MarketSpace: an Agent-Based Market Infrastructure*. In Proceedings of the 1998 Workshop on Agent-Mediated Electronic Trading. Springer-Verlag.

Guttman, R., Moukas, A. & Maes, P. 1998; *Agent-mediated Electronic Commerce: A Survey*. Knowledge Engineering Review.

Guttman, R. & Maes, P. 1998, *Cooperative vs. Competitive Multi-Agent Negotiations in Retail Electronic Commerce*. In Proceedings of the Second International Workshop on Cooperative Information Agents (CIA'98).

Hirsbrunner, B., Aguilar, M. & Krone, O. 1994, *CoLa: A Coordination Language for Massive Parallelism*. In Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC), Los Angeles, California.

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

Kielmann, T 1997, *Objective Linda: A Coordination Model for Object-Oriented Parallel Programming*. PhD thesis, Dept. of Electrical Engineering and Computer Science, University of Siegen, Germany.

Krone, O., Chantemargue, F., Dagaeff, T. & Schumacher, M. 1998, *Coordinating Autonomous Entities with STL*. The Applied Computing Review, Special issue on Coordination Models Languages and Applications.

Lamine, S.B. , Plaice, J. & Kropf, P. 1997, *Problems of computing on the WEB.* In Tentner, A. (ed), High Performance Computing, 296-301, Atlanta, Georgia, USA.

Malone, T.W. & Crowston, K. 1994, *The Interdisciplinary Study of Coordination.* ACM Computing Surveys, 26(1):87-119.

Papadopoulos, G.A. & Arbab, F. 1998, *Coordination Models and Languages*. In Zelkowitz, M. (ed), Advances in Computers, The Engineering of Large Systems, volume 46. Academic Press.

Rawston, A. & Wood, A. 1997, *Bonita: A Set of Tuple Space primitives for Distributed Coordination*. In Sprague Jr., R. H. (ed), Proceedings of the 30th Hawaii International Conference on System Sciences, Vol. 1, Wailea, Hawaii. IEEE. Minitrack on Coordination Languages, Systems and Applications.

Rodriguez-Aguilar, J. A., Noriega, P., Sierra, C. & Padget, J. 1997, *Fm96.5 a Java-based Electronic Auction House*. In Proceedings of the Second International Conference on The Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM'97).

Schubiger, S. & Krone, O. 1998, *Interactive Resource Sharing on the Web*. In Proceedings of Workshop on Distributed Computing on the WEB, Rostock, Germany, June 22-23.

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

Sun Microsystems, Inc. 1998, *Java Space TM Specification*, Revision 1.0.

Wang, D. 1999, *The Market Maker*. MIT Media Lab, Software Agents Group. Available: http://ecommerce.media.mit.edu/maker/maker.htm.

Ziemke, T. 1997, *Adaptive Behavior in Autonomous Agents*. Autonomous Agents, Adaptive Behaviors and Distributed Simulations' Journal.