# The STL++ Coordination Language: a Base for Implementing Distributed Multi-Agent Applications. *

Michael Schumacher, Fabrice Chantemargue, Béat Hirsbrunner

University of Fribourg, Computer Science Department, PAI group
Ch. du Musée 3, CH-1700 Fribourg, Switzerland
URL: http://www-iiuf.unifr.ch/pai
Email: FirstName.LastName@unifr.ch

**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 formalize communication and coordination in distributed multi-agent applications. STL++, as such, may be seen as a preliminary agent language which allows the organizational 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. STL++ uses an encapsulation mechanism as its primary abstraction, offering structured separate name spaces which can be hierarchically organized. Agents communicate anonymously within and/or across name spaces through connections, which are established by the matching of the communication interfaces of the participating agents. As an example, STL++ is used to simulate the automation of a trading system.
**Keywords**: Coordination, Distributed Systems, Concurrency, Communication Models, Agents, Multi-Agent Systems.

## 1 Introduction

Coordination can be defined as the process of *managing dependencies between activities* [29], or, in the field of Programming Languages, as the *process of building programs by gluing together active pieces*. To formalize and better describe these interdependencies, Gelernter and Carriero, in [9], propose to separate the two essential parts of a parallel application namely, *computation* and *coordination*. Because these two parts usually interfere with each other, the semantics of distributed applications is difficult to understand.

Gelernter and Carriero also state that a coordination language is orthogonal to a computation language and forms the *linguistic embodiment of a coordination*

* This work is financially supported by the Swiss National Foundation for Scientific Research, grant 20-05026.97.

*model*. Linguistic embodiment means that the language must provide language constructs either in form of library calls or in form of language extensions as a means to materialize the coordination model. Orthogonal to a computation language means that a coordination language extends a given computation language with additional functionalities which facilitate the implementation of distributed applications.

The most prominent representative coordination language is LINDA [8], which is based on a *tuple space abstraction* as the underlying coordination model. An application of this model has been realized in PIRANHA [7] (to mention one of its numerous applications) where LINDA's tuple space is used for networked based load balancing functionality. The PAGESPACE [15] effort extends LINDA's tuple space onto the World-Wide-Web and BONITA [34] addresses performance issues for the implementation of LINDA's `in` and `out` primitives. Some research tackles security issues of tuple spaces [38], [31]. Other models and languages are based on *control-oriented approaches* (IWIM/MANIFOLD [2], [3], CONCOORD [22], DARWIN [28]), *message passing paradigms* (ACTORS [1]), *object-oriented techniques* (OBJECTIVE LINDA [25], JAVASPACES [37]), *multi-set rewriting schemes* (BAUHAUS LINDA [11], GAMMA [4]) or *Linear Logic* (LINEAR OBJECTS [5]). A good overview on coordination models and languages can be found in [33].

Our work takes inspiration from control-oriented models and tuple-based abstractions, and focuses on coordination for purpose of Multi-Agent System (MAS) distributed implementations. This paper presents STL++, our C++-based coordination language. STL++ is an instantiation of the ECM coordination model which is a model for multi-grain distributed applications. For the time being, STL++ is built on top of PT-PVM [27], a software layer providing rich message passing facilities for light-weight processes (threads) over a LAN of general purpose workstations. STL++ can be considered as a platform which allows the organizational structure or architecture of a MAS to be described, with means to dynamically reconfigure it. It is conceived as a base for further multi-agent platforms. The rest of this paper is organized as follows. Section 2 examines the question of coordination in the field of MAS. Section 3 introduces the ECM model and gives a thorough description of STL++. Section 4 illustrates the application of STL++ to simulating the automation of a trading system. Section 5 is dedicated to a discussion of ECM and STL++. In the last section we draw some conclusions and outline future work.

## 2 Coordination in Multi-Agent Systems

As indicated by its name, a MAS is a macro system comprising multiple agents, each of which being a micro system. Numerous and various definitions on the notion of agent exist. Our aim here is not to cover all of them, but rather to introduce what we consider the most essential features of an agent for our concern. As said before, an agent is a system, which is situated within an environment: it senses in that environment and acts autonomously on it over time and in particular it is likely to exchange information with other agents. For more thorough

definitions, refinements and variants, see [17] and [40]. Apart from defining what an agent is and what it is aimed at, communication between agents and hence coordination are given a lot of concern. This is the object of the subsequent sub-section.

### 2.1 Communication and Coordination between Agents

Communication between agents can be considered as composed of: i) the capacity to exchange information with other agents; ii) the intention or the type of message, which is often realized using illocutary speech acts [36] such as *request*, *deny* or *confirm*; iii) a common syntax for expressing the information exchanged; iv) and a common understanding of a message. The later can be achieved if agents that participate in the communication share a common information model (which is often referred to as ontology [20] or ontological commitments).

Research has resulted in several agent communication languages (ACL). The most representative is KQML [16], which is the de facto ACL standard. KQML comes along with KIF (Knowledge Interchange Format [19]) which provides a syntax for message content along with ontologies.

Exchange of information between agents can be realized at least with three basic paradigms: i) *Peer-to-peer communication*: messages are sent to specific agents; ii) *Multicast communication*: a message is sent to a group of agents; iii) *Generative communication*: communication is realized through the environment: agents generate persistent objects (messages) in the environment to be sensed by other agents.

An agent can communicate: i) *explicitly* by having an identifier of its partner of communication; or ii) *anonymously* by putting and getting messages on well defined ports (communication interfaces); these ports are connected to other ports belonging to other agents. This means that an agent has no identifier of other agents. For anonymous communication, connections between ports can be achieved by an external specialized agent (a coordinator agent, like in [2]) or as a result of the matching of ports which depends on port characteristics.

Distributed Artificial Intelligence (DAI) has developed several techniques for coordination [23], such as *Organizational Structuring* techniques (e.g. [39]), *Multi-Agent Planning* techniques (e.g. [24]) and *Negotiation* techniques ([6]).

Multi-agent planning and negotiation techniques are designed for applications that encompass the exchange of high-level information such as plans, knowledge, beliefs or intentions. Thus they suppose basic coordination involving the communication topology between agents, which constitutes the base of more complex strategies. Organizational structuring techniques try to provide this base by supplying an a priori organization by long-term relationships between agents. This technique has shown good results, especially with master/slave or client/server patterns, sometimes using blackboard architecture. STL++ tries to resolve weaknesses encountered in organizational structuring, especially by offering means for dynamical reconfiguration, and by using a locality principle for the management of agents (in blops), thus avoiding centralization. STL++ also

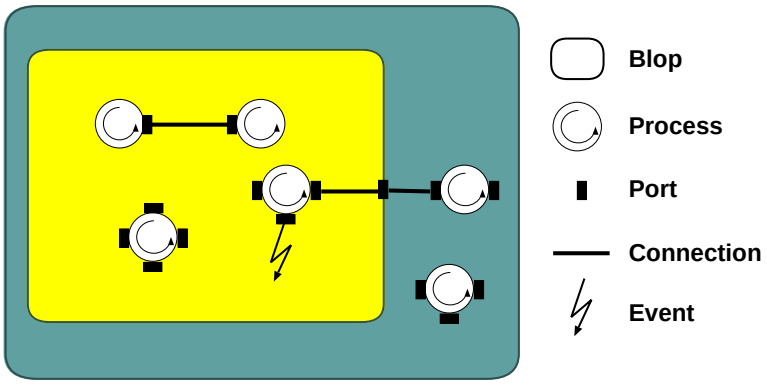


**Fig. 1.** The Coordination Model of ECM.

offers a complete set of communication primitives, that support peer-to-peer, group and blackboard communication.

## 3 STL++, a Coordination Language for Multi-Agent Systems

STL++ is a C++-based language binding of the ECM[1] coordination model [26], presented in the next section.

### 3.1 Coordination using Encapsulation: ECM

ECM uses an encapsulation mechanism as its primary abstraction (referred to as blops[2]), offering structured separate name spaces which can be hierarchically organized. Within them, active entities communicate anonymously within and/or across blops through connections, established by the matching of the communication interfaces of these entities.

ECM consists of five building blocks (see figure 1): i) *Processes*, as a representation of active entities; ii) *Blops*, as an abstraction and modularization mechanism for a group of processes and ports; iii) *Ports*, as the interface of processes/blops to the external world; iv) *Events*, a mechanism to support dynamicity (creation of new process or blop) inside a blop; v) *Connections*, as a representation of connected ports.

**Blop.** A blop is a mechanism to *encapsulate* a set of objects. Objects residing in a blop are by default only visible within their "home" blop. Blops have the same interface as processes, a (possibly empty) set of ports, and can be hierarchically structured. Blops serve as a separate name space for port objects, processes, and subordinated blops as well as an encapsulation mechanism for events.

[1] ECM stands for **E**ncapsulation **C**oordination **M**odel.
[2] This is a term we have coined.

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

**Ports and Matching.** Ports are the interface of processes and blops to establish connections to other processes/blops. Each ECM language binding must specify a minimal set of port *features*, each of which describing a port characteristics. The communication feature is mandatory and must support the following communication paradigms: point-to-point stream communication (with classical message-passing semantics), closed group (with multicast semantics) and blackboard communication. Additional features are available to refine the semantics of the communication between ports. The set of feature values of a port defines its *type*. A port is created with one or several *names*. Names are not to be used for identification, but for matching purposes. Names and type of a port are referred to as its *signature*.

The matching of ports is defined as a relationship between port signatures. Four general conditions must be fulfilled for two ports to match: i) both share at least a common name; ii) both belong to the same level of abstraction; iii) both belong to different objects (process or blop); and iv) both types must be compatible: a compliance relationship must be defined for every feature in the ECM instance. For the communication feature, ECM imposes that both ports have the same communication paradigm. Section 3 presents the features used for STL++ and the compliance relationship for each of them. The matching of ports is automatically established. This means that there exists no language construct to bind ports in order to establish a connection: the matching is therefore implicitly realized.

**Connections.** The matching of ports results in the following connections: i) *Point-to-point Stream*: `1:1`, `1:n`, `n:1` and `n:m` communication patterns are possible; ii) *Group*: messages are broadcast to all members of the group. A closed group semantics is used, i.e. processes must be members of the group in order to distribute or receive information in it; iii) *Blackboard*: messages are placed on a blackboard used by several processes; they are persistent and can be retrieved more than once in a sequence defined by the processes.

**Events.** Events can be attached to conditions on ports of blops or processes. They are typically aimed at creating new processes. The conditions will determine when the event will be triggered in the blop. Condition checking is implementation dependent.

### 3.2 The Coordination Language STL++

We designed and implemented a first language binding of the ECM model, called STL[3] [26]. STL is applied to multi-threaded applications on a LAN of UNIX workstations. STL materializes the separation of concern as it uses a separate language exclusively reserved for coordination purposes and provides primitives which are used in a computation language to express interactions between the entities. The implementation of STL is based on PT-PVM [27], a library providing message passing and process management facilities at thread and process level for a cluster of workstations. In particular, blops are implemented as heavy-weight UNIX processes, and processes as light-weight processes (threads).

However, it turned out that the separation of *code* can not always be easily maintained. Although the black box process model of ECM is a good attempt to separate coordination and computation code, dynamic properties proved to be difficult to be expressed in a separate language. This is, for example, reflected in STL, where coordination primitives have to be present in the computation language, so as to offer dynamic coordination facilities; dynamic properties can not be totally separated from the actual program code. A duplication of code is therefore inevitable and, as a result, it may introduce some difficulties to manage a distributed application. Each process and its ports must indeed be declared both in the computation language and in the separate coordination language. These observations led us to the development of a new coordination language, called STL++. Starting from the experience acquired with STL, we adopted a single language approach: STL++ implements the conceptual model of ECM by enriching a given object oriented language (C++) with coordination primitives, realized in a library.

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

**Blops.** A blop inherits from the base class `Blop` and has to re-implement the `start` method. The user must call `Blop::start()` in order to initialize the blop. The creation of a blop results in the initialization of all enclosed blops, ports and agents (STL++ instantiations of ECM processes). Blops can be parameterized like normal objects.

A blop handles all its enclosed entities; thus, a blop creates agents as well as events, and binds events to its ports. In STL++, new blops can be dynamically created at runtime.

**Agents.** Agent classes inherit from the base class `Agent` and have to re-implement the `start()` method. Agents can be created directly by blops, through events or by other agents. An agent is initialized with `Agent::start()`. To communicate, agents dynamically create ports of predefined types through instantiation of a port C++-template class.

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

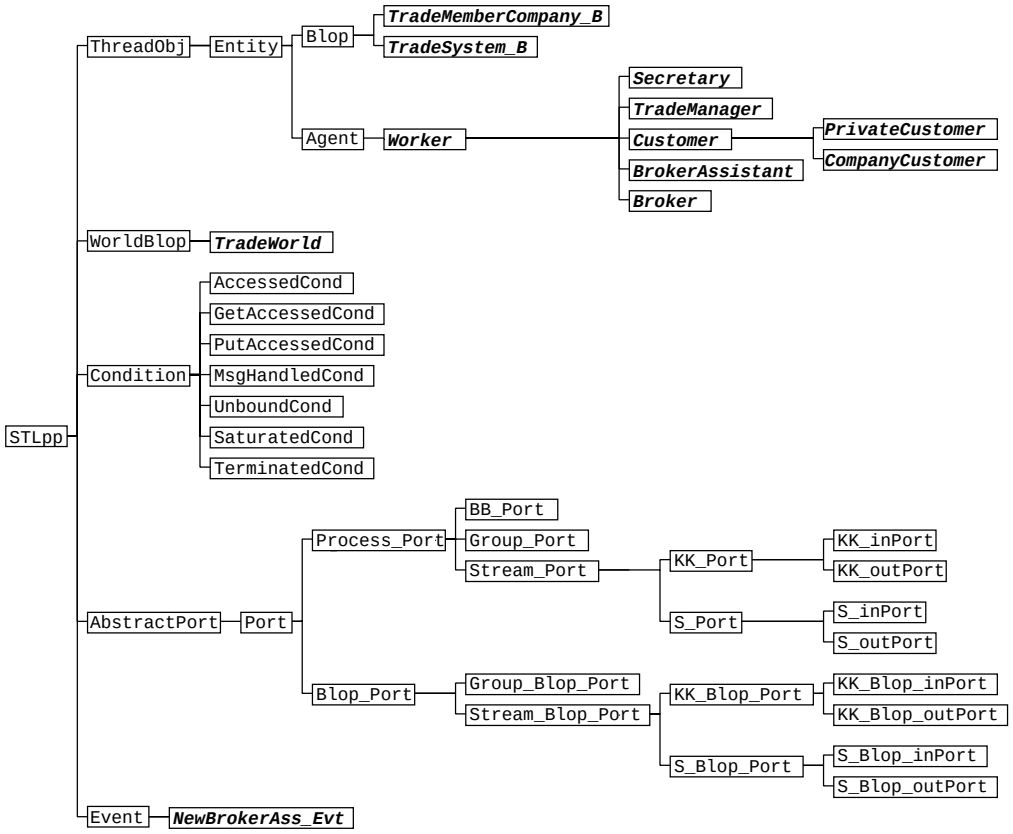


**Fig. 2.** STL++ user classes together with classes of the application presented in section 4 (bold italic).

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

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

*Primary Features*, which define the main semantics of a port, and therefore the basic port types, encompass: i) `Communication`: this feature captures ECM communication paradigms; ii) `Msg Synchronization`: this feature gives to the connection the usual semantics of message passing communication. Possible values are `synchron` and `asynchron`; iii) `Orientation`: this feature defines the direction of the data flow over a connection; there are three possible values, namely `in` (in-flowing), `out` (out-flowing) and `inout` (bi-directional).

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

**Table 1.** Basic port types and values of their primary features.

*Secondary Features*, which define characteristics for specific types of ports, are: i) `Saturation`: this feature, ranging from 1 to `INF` (infinity) by integer values, defines the number of connections a port can have; ii) `Lifetime`: this feature, ranging from 1 to `INF` (infinity) by integer values, indicates the number of data units that can pass through a port before it decays; iii) `Data Type`: this feature defines the type of data authorized to pass through a port.

STL++ currently supports four basic port types, corresponding to the combinations of the primary features as displayed in Table 1. Note that, provided that ports match, it yields four basic connection types.

`Blackboard` **port type**. The resulting connection, as a result of the matching of ports of this type, has a blackboard semantics. The number of participating ports is unlimited. Messages are persistent objects which can be retrieved using a symbolic name. Moreover, messages are not ordered. To access the blackboard, the following LINDA-like primitives are provided: `put` (non-blocking), `get` (blocking) and `read` (blocking). Blackboard connections are persistent: if all the ports involved in a blackboard connection disappear, the connection still persists in the blop space with all the information it carries, so that new ports can later on reconnect to the blackboard and recover the pending information.

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

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

`KK-Stream` **port type**. The semantics of a connection resulting from the matching of two KK-Stream ports (K for keep) is analogous to the asynchronous KK-Channel defined in [2], with its 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

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

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

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. The primitives for accessing the port are `get` (blocking) and `put` (non-blocking).

**Establishing Connections between Agents.** Communication between agents is realized through connections which are the result of matched ports. In accordance with the ECM model, the matching is realized as a relation between port signatures. In STL++, in order to match, ports must belong to the same blop and must comply at name and type levels: i) *Name level.* Two ports match at name level if they share at least one name. A port may have several names; in this case, each name belongs to a different connection; ii) *Type level.* For two ports, values of the same feature must be compatible. Table 2 gives an overview of the compatibility functions used by the STL++ runtime system.

By introducing several names for each port, STL++ allows stream and group ports to be connected to different connections (blackboard ports cannot have multiple connections). Data written on such multiple connection ports are echoed on every connection. For stream connections, `1:1`, `1:n`, `n:1` and `n:m` communication patterns can be built. Likewise, several groups can be connected to a single port.

**Events.** Event classes inherit from `Event`; the `launch()` method, which defines the acting of the event, must be re-implemented. Events are instantiated with a specific lifetime which determines how many times they can be triggered.

Conditions on ports (see table 3 for an overview) are checked when data flow through the port, or, in the case of `saturatedCond` condition, when a new connection is realized.

## 4 Simulation of a Trading System

The numerous activities that take place within a trading system are typically distributed and can be modeled by a multi-agent system. It has led solutions

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

**Table 3.** Port Conditions

for agent-based electronic commerce; see for instance KASBAH [13], FISHMARKET [35], or a proposal based on the PAGESPACE platform [14]. If our goal is to fully automate a trading system, for the time being, we would 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. [21]) 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 [12] in the sense that they can make decisions on their own, without user intervention. More sophisticated autonomy-based control algorithms and smart negotiation techniques will be tackled in a further stage.

Figure 3 gives a scaled down graphical overview of the organization of the agents that compose our trading system, as well as their interactions. To avoid cluttering 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 authorized members of a trading system that represent 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 whom is devoted to serve a particular customer, by 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` in figure 3), 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 `NewBrokerAss_Evt` with the condition `PutAccessedCond`; the effect of the posting is that the `NewBrokerAss_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 assis-

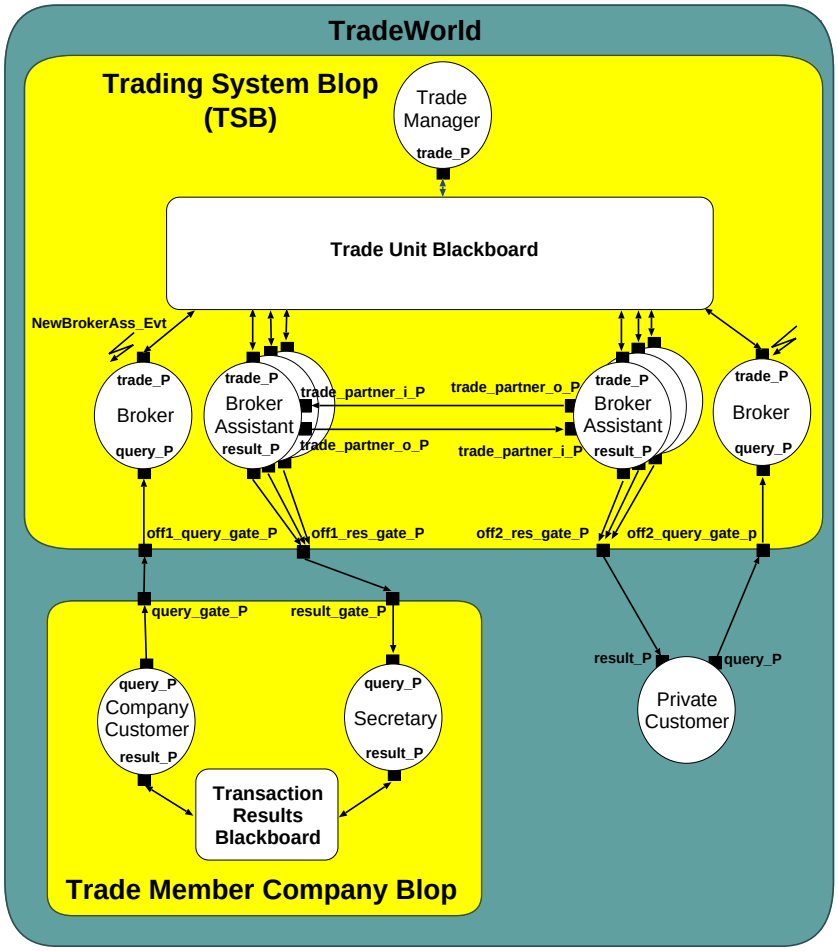


**Fig. 3.** Trading System with STL++.

tant 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). If 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 newly created Broker Assistant has first to read from his `trade_P` port (of type `BB_Port`) in order to be informed of a specific transaction. In virtue of 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 (the 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 3). At regular intervals, non fulfilled queries are eliminated by the Trade Manager; all involved entities are kept posted.

## 5 Discussion

### 5.1 STL++ a Coordination Language

ECM, a coordination model for distributed programming, along with STL++ its instantiation, present some similarities with e.g. the IWIM [2] coordination model and its instance MANIFOLD [3], LINDA [8], DARWIN [28] or CONCOORD [22].

Though ECM shares many characteristics with IWIM, it however differs in several points: i) Blops are not coordinators like IWIM managers. Connections are not established explicitly by the blop (or by another agent), i.e. there is no language construct to bind ports in order to establish a connection: the establishment of connections is done implicitly, resulting from a matching mechanism between compatible ports within the same blop; the matching depends on the types and the states of the concerned ports. The main characteristics of blops is to encapsulate objects, thus forming a separate name-space for enclosed entities and an encapsulation mechanism for events; ii) ECM generalizes connection types, namely stream, blackboard or group, and does not restrict to channels; iii) In ECM, events are not signals broadcast in the environment, but objects that realize an action in a blop. Events are attached to ports with conditions on their state that determine when they are to be launched. They can create agents or blops, and their action area is limited to a blop.

ECM together with STL++ differ in the following points from LINDA: i) Like several further developments of the LINDA model (e.g. [25]), ECM uses a hierarchical multiple coordination space model [18], in contrast to the single flat tuple space of the original LINDA. But it has to be stressed that a blop is not a shared dataspace as a tuple space, but an abstraction mechanism of several agents that serves as a separate name space; ii) In STL++ agents get started through an event in a blop, or automatically upon initialization of a blop, or through a creation operation by another agent; LINDA uses a single mechanism:

`eval()`. The termination of an agent results in the loss of all its enclosed data and ports; it may as well result in the loss of pending data in the case of unbound connections. (see section 3). In Linda, a tuple replaces the terminated process; iii) In STL++ agents do not execute in a medium which is used to transfer data. Every communication is realized through connections established by the matching of ports. Only the engaged agents have access to those connections; iv) Blackboard connections allow for generative communication between the participating agents. For the time being, a string is used to introduce and retrieve data. Neither tuples, nor templates, nor pattern matching are supported for retrieving data. Blackboards are typed in the sense that only one type of data can be entered. There is also no kind of active tuples.

### 5.2 Implementing Coordination in MAS with STL++

STL++ can be considered as a preliminary language for constructing agent applications, stressing the coordination part of a MAS. In its purpose, STL++ is comparable to APRIL [30], conceived as a platform oriented towards the implementation of MAS, and to BAUHAUS LINDA [10], which has been applied to implementing Turingware, a superset of Groupware.

STL++ has some characteristics that make it a good candidate for implementing MAS: i) Blops constitute a mechanism which enables construction of a structured set of different environments, each of which is a closed private coordination space with communication interfaces to other environments. A set of complex interconnected and hierarchically organized spaces is indeed often desirable in MAS; ii) Autonomous agents are implemented as black boxes. An autonomous agent owns exclusive control over its internal state and behavior; 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. iii) Sensing and acting capacities of agents are implemented with ports. Thus agents can perceive and act through different means, by 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. iv) As agents and environments may evolve over time, dynamicity is an important point in MAS. This has led to design STL++ so as to support blop reorganization, new port creation and destruction, thus yielding dynamic communication topologies.

## 6 Conclusion

In résumé, we presented the ECM coordination model and STL++, an instantiation that constitutes a platform to implement Multi-Agent Systems. An STL++ implementation of a simulation of a Trading System showed the potential of our coordination language.

Future work will consist of: i) Extending STL++ so as to support generic coordination patterns, yielding templates at disposal for general purpose implementations; ii) Enhancing the Trading System simulation so as to tackle more sophisticated autonomy-based control algorithms and smart negotiation techniques. We also work on a new ECM mapping in Java built on the ORBacus Object Request Broker [32] and the JavaSpaces [37] API.

## References


1. G. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.
2. F. Arbab. The IWIM Model for Coordination of Concurrent Activities. In P. Ciancarini and C. Hankin, editors, *Proceedings of the First International Conference on Coordination Models, Languages and Applications*, number 1061 in LNCS. Springer Verlag, April 1996.
3. F. Arbab, I. Herman, and P. Spilling. An Overview of Manifold and its Implementation. *Concurrency: Practice and Experience*, 5(1):23–70, February 1993.
4. J.P. Banâtre and D. Le Métayer. Programming by Multiset Transformation. *Communications of the ACM*, 36(1):98–111, 1993.
5. M. 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. S. Bussmann and J. Muller. A Negotiation Framework for Co-operating Agents. In S. M. Deen, editor, *Proceedings of CKBS-SIG*, pages 1–17. Dake Centre, University of Keele, 1992.
7. N. Carriero, E. Freeman, D. Gelernter, and D. Kaminsky. Adaptive Parallelism and Piranha. *IEEE Computer*, 28(1), January 1995.
8. N. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, 1989.
9. N. Carriero and D. Gelernter. Coordination Languages and Their Significance. *Communications of the ACM*, 35(2):97–107, February 1992.
10. N. Carriero, D. Gelernter, and S. Hupfer. Collaborative Applications Experience with the Bauhaus Coordination Language. 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.
11. N. Carriero, D. Gelernter, and L. Zuck. Bauhaus Linda. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *Lecture Notes in Computer Science*, Berlin, 1995. Springer Verlag.
12. F. Chantemargue, O. Krone, M. Schumacher, T. Dagaeff, and B. Hirsbrunner. Autonomous Agents: from Concepts to Implementation. In *Proceedings of the Fourteenth European Meeting on Cybernetics and Systems Research (EMCSR'98)*, volume 2, pages 731–736, Vienna, Austria, April 14-17 1998.
13. A. Chavez and P. Maes. 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, April 1996. Lawrence Erlbaum.

14. P. Ciancarini, A. Knoche, D. Rossi, R. Tolksdorf, and F. Vitali. Coordinating Java agents for Financial Applications on the WWW. In *Proceedings of the Second International Conference and Exhibition on The Practical Application of Intelligent Agents and Multi-Agents, PAAM '97*, 1997.
15. P. Ciancarini, A. Knoche, R. Tolksdorf, and Fabio Vitali. PageSpace: An Architecture to Coordinate Distributed Applications on the Web. In *Proceedings of the Fifth International World Wide Web Conference*, volume 28 of *Computer Networks and ISDN Systems*, 1996.
16. T. Finin, R. Fritzon, D. McKay, and R. McEntire. KQML as an Agent Communication Language. In *Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM)*. ACM Press, 1994.
17. S. Franklin and A. Graesser. Is it an Agent or just a Program? A Taxonomy for Autonomous Agents. In J.P. Muller, M.J. Wooldridge, and N.R. Jennings, editors, *Proceedings of ECAI'96 Workshop (ATAL). Intelligent Agents III. Agent Theories, Architectures, and Languages*, number 1193 in LNAI, pages 21–35, August 1996.
18. D. Gelernter. Mulitiple Tuple Spaces in Linda. In E. Odijk, M. Rem, and J.Syre, editors, *Proceedings of the Conference on Parallel Architectures and Languages Europe (PARLE 89)*, volume 365 of *Lecture Notes in Computer Science*, pages 20–27, Berlin, 1989. Springer Verlag.
19. M.R. Genesereth and R.E. Fikes. Knowledge Interchange Format, Version 3.0. Reference Manual. Technical Report Logic-92-1, Computer Science Department, Stanford university, 1992.
20. T.R. Gruber. A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, 5:119–220, 1993.
21. R. Guttman and P. Maes. Cooperative vs. Competitive Multi-Agent Negotiations in Retail Electronic Commerce. In *Proceedings of the Second International Workshop on Cooperative Information Agents (CIA'98).*, July 1998.
22. A.A. Holzbacher. A Software Environment for Concurrent Coordinated Programming. In P. Ciancarini and C. Hankin, editors, *Proceedings of the First International Conference on Coordination Models, Languages and Applications*, number 1061 in LNCS. Springer Verlag, April 1996.
23. L. Lee H.S. Nwana and N.R. Jennings. Co-ordination in Multi-Agent Systems. In H. S. Nwana and N. Azarmi, editors, *Software Agents and Soft Computing*, number 1198 in LNAI. Springer Verlag, 1997.
24. Y. Jin and T. Koyoma. Multi-agent Planning through Expectation-based Negotiation. In *Proceedings of the 10th Int Workshop on DAI*, Texas, 1990.
25. T. Kielmann. *Objective Linda: A Coordination Model for Object-Oriented Parallel Programming*. PhD thesis, Dept. of Electrical Engineering and Computer Science, University of Siegen, Germany, 1997.
26. O. Krone, F. Chantemargue, T. Dagaeff, and M. Schumacher. Coordinating Autonomous Entities with STL. *The Applied Computing Review*, Special issue on Coordination Models Languages and Applications, 1998.
27. O. Krone, B. Hirsbrunner, and V.S. Sunderam. PT-PVM+: A Portable Platform for Multithreaded Coordination Languages. *Calculateurs Parallèles*, 8(2):167–182, 1996.
28. J. Magee, N.Dulay, and J. Kramer. Structuring parallel and distributed programs. *Software Engineering Journal*, pages 73–82, March 1993.
29. T.W. Malone and K. Crowston. The Interdisciplinary Study of Coordination. *ACM Computing Surveys*, 26(1):87–119, March 1994.

30. F.G. McCabe and K.L. Clark. April - agent process interaction language. In M. Wooldridge and N.R. Jennings, editors, *Intelligent Agents. Proceedings of First International Workshop on Agent Theories, Architectures, and Languages (ATAL'94)*, number 890 in LNAI. Springer Verlag, August 1994.
31. R. De Nicola, G. Ferrari, and R. Pugliese. Coordinating mobile agents via blackboards and access rights. In *Proceedings of the 2nd International Conference on Coordination Languages and Models (Coordination '97)*, number 1282 in LNCS. Springer Verlag, September 1997.
32. ORBacus Object Request Broker. Object Oriented Concepts, Inc., 1999. www.ooc.com.
33. G.A. Papadopoulos and F. Arbab. Coordination Models and Languages. In M. Zelkowitz, editor, *Advances in Computers, The Engineering of Large Systems*, volume 46. Academic Press, August 1998.
34. 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.
35. J. A. Rodriguez-Aguilar, P. Noriega, C. Sierra, and J. Padget. 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)*, 1997.
36. J. Searle. *Speech Acts.* Cambridge University Press, 1969.
37. Sun Microsystems, Inc. *JavaSpaces TM Specification, Revision 1.0*, March 1998.
38. R. van der Goot, J. Schaeffer, and G.V. Wilson. Safer tuple spaces. In *Proceedings of the 2nd International Conference on Coordination Languages and Models (Coordination '97)*, number 1282 in LNCS. Springer Verlag, September 1997.
39. K.J. Werkman. Knowledge-based Model of Negotiation using Shareable Perspectives. In *Proceedings of the 10th Int Workshop on DAI*, 1990.
40. M. Wooldridge and N.R. Jennings. Agent Theories, Architectures, and Languages: a Survey. In M. Wooldridge and N.R. Jennings, editors, *Intelligent Agents*, number 890 in LNCS, pages 1–39. Springer Verlag, 1995.