

Coordination Technologies for Business Strategy Support: a case study in Stock Trading

G.Koutsoukos¹, T. Kotridis², L.Andrade^{1,3}, J.L.Fiadeiro^{3,5}, J.Gouveia¹ and
M.Wermelinger^{3,4}

¹ OBLOG Software S.A., Alameda António Sérgio 7, 2795 Linda-a-Velha, Portugal
{gkoutsoukos,jgouveia,landrade}@oblog.pt

² Accenture, London, UK
theofilos.kotridis@accenture.com

³ ATX Software SA, Alameda António Sérgio 7, 2795 Linda-a-Velha, Portugal

⁴ Dep. de Informática, FCT, Univ. Nova de Lisboa, 2825-114 Caparica, Portugal
mw@di.fct.unl.pt

⁵ Dep. de Informática, Fac. Ciências, Univ. Lisboa, Campo Grande, 1700 Lisboa, Portugal
jose@fiadeiro.org

Abstract. In today's global and highly competitive business environments, organizations are replying to the question of whether technology is forming business or vice-versa by integrating their business and IT strategies, thus using technology to do business. As a result, information systems are at the core of the competitive edge of every business organization, which puts an increasing pressure for endowing them with the levels of flexibility and agility that are required to support changes of business strategies and operate in what have become highly volatile business domains. In this paper, we argue in favor of the adoption of “software strategic libraries” based on the “coordination technologies” that we have been developing, in order to support “business reactive” software systems. We support our view by presenting examples from the highly volatile, and extremely competitive, stock-trading business domain.

1 Introduction

Three years ago, Hammer and Champney [10] argued that the progression of events in the New Economy follows the following pattern: “*customer takes control, competition intensifies and change becomes constant. There is an urgent need for transformation*”. Today, “transformation”, both by reengineering business processes and by making innovative use of technological advances to do business, has become a critical success factor that every organization is striving to achieve. Because, more and more, business is driven through software solutions, organizations need information systems that are very adaptive to changes, even ones performed directly by customers, and easily reconfigurable, often in run-time. Through the advent of the Internet and Wireless Applications, the New Economy is only fuelling this need even further, namely in the context of e-commerce: “... *the ability to change is now more important than the ability to create e-commerce systems in the first place. Change*

becomes a first-class design goal and requires business and technology architecture whose components can be added, modified, replaced and reconfigured” [6].

Unfortunately, while various disciplines for Business Change, namely Business Process Reengineering [10], have been proposed and put in practice over the previous years, there is still a lack of systematic and scalable solutions that can address software evolution in general and in the face of changes in the business domain in particular. As argued in some of our recent papers [1, 2, 3], OO languages and the technologies associated with component-based frameworks have fallen short of redressing this situation, which may explain why software teams are still struggling to compete with the fast business and technology evolution and, as shown by numerous scientific studies on large-scale software systems, more than 80% of the total cost of software development is still devoted to software maintenance. The main reason is that the support that OO techniques such as inheritance and clientship offer for system construction cannot be extended directly to system evolution: they are too “static” and “white-box” when it comes to change. On the other hand, design patterns offer solutions that are too low level to be able to support an evolution process that takes place at the much higher level of abstraction in which business strategies and rules are (re)defined: they are useful for providing the design infrastructure that will support the required levels of adaptability, but they cannot be used for modeling and controlling the evolution process by themselves.

In this paper we show how a new semantic primitive – coordination contract – that we presented in [2] as an extension to OO modeling languages, together with the design patterns that support its implementation over component-based platforms [4, 9], can be used in support for a new approach to Business Modeling based on the definition of “software strategic libraries” and (re)configuration mechanisms that will deliver “business reactive” information systems. We present our case on the basis of some simple, but real-life examples from the Stock-Trading industry – an industry characterised by its high volatility, in more than one sense, or, in the words of an unsuspected source, the place in which “the bloodiest financial services battle on the Internet will be” [7].

The remainder of this paper is organized as follows: Section 2 expands on these introductory remarks and discusses the ways in which coordination technologies based on contracts can improve the way strategic business solutions can be supported. Section 3 gives an overview of the basic concepts of coordination contracts. Section 4 presents a contract-based architecture of a Stock-Trading subsystem in order to illustrate how contracts can be applied in order to build systems that dynamically support different business strategies. Concluding remarks and an outline of on-going and future work round up the paper.

2 Coordination in Stock Trading

The Stock-Trading Business domain in general, and stock trading systems in particular, is very complex and very volatile. They involve a number of different people, even possibly different organizations, equipment, material resources and

business procedures. Building stock-trading systems normally involves strategic (business), social (human resources), and technological decisions. In this paper, we will try to show that a new technology is being made available that can lead to a new way in which this complexity can be tackled so that information systems can, indeed, be regarded as facilitators in the deployment of Business Strategies, i.e., as part of the solutions that organizations can rely upon, and not of the problems that they have to face, when it comes to competing in very dynamic environments. In order to do so, we consider a stock trading process as an example and focus on the variety of accounts and types of trading that are usually offered to a client.

Coordination contracts are the semantic primitives that are at the core of the “Coordination Technologies” that we have been developing for supporting Information System Evolution in such turbulent environments [1]. These technologies are based on the separation between what in systems is concerned with the computations that are responsible for the functionality of the services that they offer and the mechanisms that coordinate the way components interact, a paradigm that has been developed in the context of so-called Coordination Languages and Models [8]. The rationale of the application of coordination technologies to Business Modeling is in the realization that, in highly volatile business domains, one can usually distinguish between two different kinds of “entities” as far as the evolution of the domain is concerned. On the one hand, there are classes of entities that correspond to entities that are relatively “stable” in the sense that they capture core concepts of the business domain and, therefore, do not change very frequently or for which the organization accepts that any change may require a more global impact. On the other hand, there are entities that need to keep changing in order for the system to reflect the dynamics of the application domain, typically capturing the evolution of the business rules under which the organization is operating. These require a layer of coordination to be superimposed over the functionalities provided by the stable entities so that the global behavior that is expected from the system can emerge, at each state, from the computations performed locally in components and the interconnections that this layer of coordination puts in place among them. These coordination aspects need to be made available explicitly in system models so that they can be changed, as a result of modifications occurring at the level of requirements, without affecting the components that ensure the functionalities of the basic services of the system. The purpose of coordination contracts is to provide mechanisms for that layer of coordination to be modeled and evolved in a compositional way.

For instance, accounts and types of trading are highly volatile business assets that determine the competitive edge of a stock trading organization. Therefore, the information systems that support Stock Trading must be structured in a way that changes at the level of these aspects can be easily accommodated, even at run time, as driven through the Internet, with minimum cost and impact on the services already implemented. As an example, consider the account types that investors in stock trading are offered in order to perform trading. Most of the stock-trading firms today offer Traditional Accounts, Margin Accounts, Flexible Accounts, Discounter Accounts, Upper or Low Quantity Limit Accounts, among others. For each of them, the trading company specifies different business rules that regulate the specific forms of trading that they support. For instance, a Margin Account allows the customer to

perform stock trading by borrowing money from the firm on short sales. This allows the trader to increase buying power for a period of time with the obligation that there is enough cash in the account, for instance a minimum balance of an agreed amount, say 60000 currency units. When trading is performed using a margin account, an order can be committed only if the balance of the account plus the amount awarded by the firm (MarginLimit) is greater than, or equal to, the price of the stocks required plus the minimum balance. However, such requirements, which are also specified for all the other types of accounts, can change according to different market situations.

Clearly, the frequent and unpredictable evolution of account types, with types added, changed, or removed according to market rules, as well as the modifications of the legal rules that regulate stock trading based on such accounts, makes evolution a critical concern when designing systems that have to support services such as these. It is only natural that, given the promotion that they have received in the recent past, companies seek solutions for this problem in OO technology. However, whereas OO techniques such as Inheritance and Clientship have proved to be adequate for *constructing* business information systems, their support for the *evolutionary* aspects is not so straightforward.

On the one hand, inheritance as a means of modeling new situations is not “dynamic”. In other words, if a component needs to be modified, the use of inheritance requires us to know, understand, and modify its internals. For instance, if we model MarginAccount as a subclass of an Account type, new variations of margin accounts would require solutions that either imply modifying the MarginAccount class or creating new subclasses, which are both intrusive on the implementation of existing system components. Even worse, after a MarginAccount type is created, a future removal of such an asset from the business domain, or simply a modification of its features, cannot be easily reflected on the system implementation without triggering changes to all parts of the system in which MarginAccount participates.

On the other hand, with clientship, interactions become “hard-wired” in the code that implements the participating objects, making it difficult to change or introduce new interactions without having to change the implementation of the objects as well. Even worse, because such changes may result in new interfaces for the participating objects, a cascade of changes throughout the implementation of the system may well be triggered to account for the other interactions in which the objects participate. For instance, “hard-wiring” in an operation BuyStock of a StockOrder object the conditions (guards) under which a specific stock order for a margin account is enabled would lead to a system structure in which extensions or modifications on the code that implements the operation and its clients would be required in order to reflect new trading conditions for the account.

Hence, from the evolution point of view, volatile business requirements or assets such as Margin Trading should be modelled explicitly outside the classes and operations that model the basic business entities such as Customer, Account, Deposit or BuyStock, so they can be evolved independently of those entities. In other words, the desired separation of concerns must be available right from the earlier, more abstract system models that result from traditional analysis techniques, and carried through to the implementation level. Coordination contracts and their deployment patterns provide exactly this ability by supporting a clear separation of computation

from interaction, allowing the coordination aspects to be externalized and handled explicitly as first-class citizens.

3 Coordination Contracts

In general terms, a coordination contract is a connection that is established between a group of objects (participants). Through the contract, rules and constraints are superposed on the behavior of the participants, which determines a specific form of interaction. From a static point of view, a contract defines what in the UML is known as an *association class*. However, the way interaction is established between the participants is more powerful than what can be achieved within the UML and similar OO languages because it relies on the mechanism of *superposition* as developed for parallel program design [11]. When a call is made from a client object to a supplier object, the contract “intercepts” the call and *superposes* the forms of behavior it prescribes. In order to provide the required levels of “pluggability”, neither the client, nor any other object in the system, may know what kind of coordination is being superposed. To achieve this “black box” view of system components, a contract design pattern was developed as presented in [4, 9].

Coordination contracts are currently supported by a specification language called Oblog [14], but the underlying technology is independent of the language in the sense that the semantic primitive and the design pattern can be used in the context of other modeling languages and methods. Using the Oblog notation, a coordination contract is defined as follows:

```
contract class <name>
  participants <list of partners>
  constraints <the invariant the partners should satisfy>
  attributes
  operations
  coordination
end class
```

A contract consists of a collection of role classes that identify the types of objects that can be partners in the contract, constraints that represent invariants defining in which conditions instances from the participating classes may be related by the contract, attributes and operations private to the contract, and the prescription of the coordination effects that will be superposed on the partners.

Coordination is prescribed through a number of rules of the form:

```
<name>      when <trigger>
            with <condition>
            do <set of actions>
```

The name of the rule identifies a particular form of coordination; it identifies a point of “rendez-vous” in which the participants have to synchronize their behavior. The names themselves are used for managing the interference between different contracts that may be active in the same state as discussed further below.

For each rule, the condition under “when” identifies the trigger that prompts the contract to become active and coordinate the behavior of the participants. The trigger

can be a condition on the state of the participants, a request for a particular service, or an event on one of the participants. Several trigger conditions can be placed in the “when” clause using the keyword “AND”. If one of such conditions is not satisfied, the contract is considered as being “inactive” and, as a result, the participants progress independently of the reaction specified in the rule. This mechanism provides the ability for controlling which of the contracts imposed on a component will be responsible for coordinating it, thus allowing for dynamic configuration of the component behavior.

The “do” part of each rule identifies a synchronization set of actions of the partners and some of the contract’s own actions. This set is required to be executed atomically in the sense that if the execution of any of its actions fails, the execution of the rule itself fails.

When the trigger corresponds to the call for an operation of one of the partners, three types of actions may be superposed on the execution of the operation:

- **before** actions: to be performed before the operation
- **replace** action: to be performed instead of the operation (alternative)
- **after** actions: to be performed after the operation

In the case in which an object participates in multiple contracts with the same trigger, the sequence of execution for the before, replace and after clauses is shown in Figure 1. It should be noted that the semantics of contracts allows for only one “replace” clause to be executed, thus preventing the undesirable situation of having two alternative actions for the same trigger. Furthermore, any such replacement action

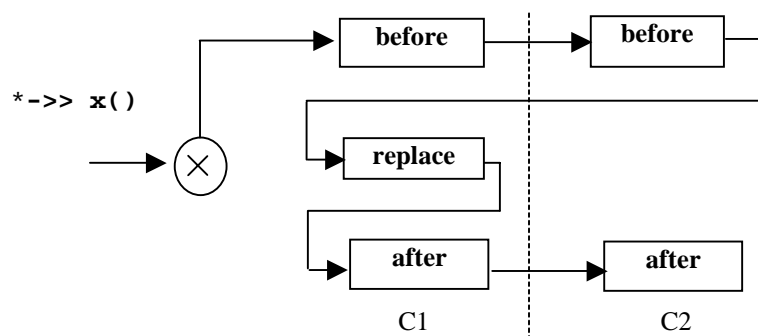


Fig.1. Execution of before, replace and after

must adhere to whatever specification clauses apply to the operation (e.g., contracts in the sense of [13] specifying pre- and post-conditions). This ensures that the functionality of the original operation, as advertised through its specification, is preserved.

Each synchronisation set is guarded by the conjunction of the guards of the individual actions together with the conditions included in the “with” clause. Therefore, the “with” clause puts further constraints on the execution of the actions involved in the interaction. If any condition under the “with” clause is not satisfied, the synchronisation set fails and none of its actions is executed.

For a more detailed description of coordination contracts and the technology that puts them in practice, the reader is urged to consult [2, 3, 4, 9]. In what follows, we present an example from the Stock Trading application domain to illustrate how contracts can be used for building systems that support evolving business strategies.

4 Contract-based solutions for Stock Trading

In order to illustrate the points made in the previous sections, consider the simple object architecture of Figure 2 that represents a stock trading subsystem with coordination contracts established between customers and stock orders. In a client-server architecture, Figure 2 represents some of the objects that constitute what we may call a “customer session” existing on the server side for each customer logging on the client side. Note that the architecture of Figure 2 is only an adaptation of a real-life architecture made in order to illustrate the use of contracts. Therefore, it omits details irrelevant to contracts like support for performance optimization.

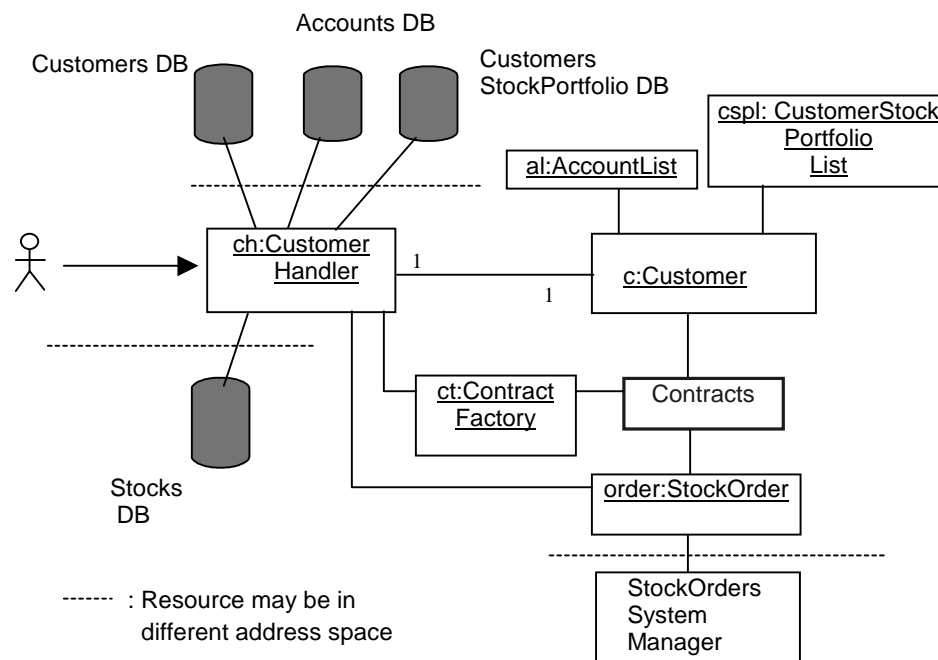


Fig. 2. Customer-session server objects of a contract-based stock trading subsystem

The functionality of the classes of objects involved is straightforward. The CustomerHandler is responsible for communicating with the client side and retrieving the required resources from the various Databases (DBs). A Customer object represents the customer logging on the trading system, and maintains the list of

Accounts and the list of the StocksPortfolio for the Customer. A StockOrder object is created at the back-end to model the order created by the actor at the front-end. A typical StockOrder object may contain attributes such as orderNumber, StockSymbol, desiredPrice, quantity and so on, which model the stock orders taking place in the business domain.

The object can communicate with a StockOrdersSystemManager that commits transactions or updates DBs according to the system architecture. Notice that it is also possible to use contracts to regulate such communications but presenting such contracts in any meaningful level of detail is not in the scope of this paper.

A ContractFactory is an optional object for creating the contracts in place for a particular Customer and a particular StockOrder. We are currently working on a configuration language for contracts that will address issues like that but its description is, again, outside the scope of this paper. Our concern is to show how, by using contracts, volatile interactions between objects can be independently modeled and implemented, thus providing the opportunity for building systems that that can be easily modified.

Consider the scenario in which a client wants to buy a number of Stocks using the concept of a Margin Account. After the StockOrder is created (both at the front and back-end) the order is pre-submitted to the system so that the conditions required for the order to be valid can be checked. For instance, as described before, it is necessary to check whether the funds in the customer's account satisfy the conditions for Margin Trading and act accordingly, either accepting or rejecting the order. The following contract between a Customer and a StockOrder, models the business rules related to a MarginAccount trading.

```

contract MarginAccountTrading
participants
  c:Customer;
  order:StockOrder;
attributes int MarginLimit=30000,MinimumBalance=60000
coordination
  marginTrading:
    when*->order.Buy(accountNumber,stockSymbol,quantity,StockPrice)
    AND (c.getAccount(accountNumber).getBalance())>=MinimumBalance)
    with (c.getAccount(accountNumber).getBalance()+MarginLimit
          >=(quantity*StockPrice+MinimumBalance))
    replace{
      order.Buy(accountNumber,stockSymbol,quantity,StockPrice);
    };
end contract //MarginAccountTrading

```

The contract models the characteristics of a MarginAccount by including private attributes such as MarginLimit and MinimumBalance. In a traditional OO approach, a MarginAccount concept would be considered as a special type of Account. However, this account is more related to trading rules than to the usual core business domain entity Account. In other words, from the evolution point of view, it makes more sense to model the functionality of the system by considering the business concept of a MarginAccount as a trading type rather than as a core business entity. The use of

coordination contracts dispenses the use of inheritance for modeling situations like these, thus avoiding the problems that we mentioned when new concepts are added to that business entity. Instead, new attributes or behavior can be superposed dynamically on the corresponding objects through relevant contracts as illustrated above. At the same time, the previous contract provides another important advantage: the externalization of the business rules that regulate MarginTrading. In other words, the volatile business rules that determine the conditions under which MarginTrading is allowed, even if they are different for specific customers or orders, can be modeled right from the analysis phases and implemented in a way that can be changed without affecting the functionality of the rest of the system. For instance, if a strategic decision requires new rules to be related to MarginTrading, a new contract can be inserted to the system in a “plug and play” mode to support this decision without having to “touch” the basic objects that compose the system.

Consider now the case in which the trading firm decides to introduce a new type of trading in order to attract new customers. In fact, this was the case of e-stock trading firms such as Charles Schwab [15] that introduced the notion of Discounter Account trading, a type of trading with a very low fee. In Discounter Account trading, the customer is required to have an account balance between a minimum and a maximum amount, and the balance must be greater than the sum of the total price of stocks ordered plus the minimum amount plus the trading fee. This new strategic type of trading can very easily implemented and plugged to the system using a contract such as the one below:

```

contract DiscounterTrading
participants
  c:Customer;
  order:StockOrder;
attributes  int MinimumBalance=40000, MaximumBalance=70000;
             double tradingFee=20;
operations String getMonth();
coordination
  discounterTrading:
    when*->>order.Buy(accountNumber,stockSymbol,quantity,StockPrice)
    AND (c.getAccount(accountNumber).getBalance()>=MinimumBalance
        &&  getAccount(accountNumber).getBalance()<=MaximumBalance
        &&  getMonth()=="June")
    with (c.getAccount(accountNumber).getBalance()
        >=(quantity*StockPrice+MinimumBalance+tradingFee)
    replace{
      order.Buy(accountNumber,stockSymbol,quantity,StockPrice);
    };
end contract //DiscounterTrading

```

Notice that we may configure the contract to be active only when the month is, for instance, June. This kind of configuration allows for implementing short-term or long-term strategies corresponding to decisions occurring at the level of business requirements. Clearly, similar contracts can be specified for FlexibleTrading, UpperQuantityLimitTrading or any other type of trading the management would like to introduce in order to support the goals of a new business strategy.

Apart from the previous examples, there is another capability of contracts that is interesting to discuss: to have state conditions as triggers. Consider, independently of the architecture of Figure 2, the following contract that may be part of an “intelligent” stock trading system. The contract performs an automatic Buy action for a Customer’s Account when the price of a Stock is greater than a buying threshold. Naturally, similar “intelligent” contracts may be specified for other business rules such as “Selling High”. Such contracts, which may be specified directly by the customer or a trader, may allow companies to implement different management decisions, reduce costs and attract new customers.

```

contract BuyLowContract
participants    stock: Stock; account: Account;
                //other necessary system participants
attributes double BuyMargin =0.50, int quantity=2000;
coordination
  BuyLow:
    when ?(stock.getLastPrice()-stock.getPrice()> BuyMargin)
    do {
      StockOrder order=new StockOrder(ordernumber, stock, quantity, lastStockPrice);
      order.Buy(account.number, stock.stockSymbol, quantity, order.Price);
    };
end contract

```

The nature of the triggers that can be used in coordination rules depends, ultimately, of the languages and platforms in which system components are programmed and deployed. As already mentioned, coordination technologies are, essentially, language and platform independent in the sense that the underlying principles like coordination and superposition are “universal”. However the degree of coordination that can be achieved will always depend on the mechanisms that are offered for components to interact, for error recovery, for transaction management, etc.

4 Concluding Remarks

In today’s global and highly competitive business environments, to the question of whether technology is forming business or vice-versa, organizations are replying by integrating their business and IT strategies [5], thus using technology to do business. Online business and virtual organizations is the trend as E-commerce numbers are increasing and the emerging wireless data technologies are fuelling the creation of new business opportunities. Flexibility, innovation and change have become critical success factors for every business organization. Business strategies have now, more and more, a short-term time horizon and, for the first time, can be put in practice by information systems alone (strategic information systems). As a consequence, there is an increasing pressure for building software systems that are dynamically reconfigurable and adaptive to changes imposed either by technology innovation or new business needs.

In this paper, we presented the technologies that, in our opinion, and from the experience that we have gathered in the financial and telecommunications sectors, can open the way to a new approach to building software systems that addresses the dynamics of business strategies. We showed how a modeling primitive, named coordination contract, can put in practice to structure software systems into two layers: The computation layer, consisting of all static components that do not encapsulate any business logic and do not evolve over large periods of time; and the coordination layer, consisting of those components that describe the business logic and therefore are subject to evolution. In other words, our approach promotes the idea of building systems using libraries of “stable” components and “strategic evolving” components. Borrowing examples from the stock-trading business domain, we showed how such system structuring and component libraries can assist management and marketing teams to put in practice different kind of strategies, aggressive or defensive, short term or long term, and so on.

We are also convinced that there are a large number of application domains in which contracts can be applied in order to support software evolution imposed by such strategic decisions. However, there is no doubt that there is a lot to be done before these technologies can find their way into the business world as effective modeling and development solutions. Therefore, we are currently working on different aspects of the development methodology associated with coordination contracts, namely in what concerns distribution, configuration, and the modeling of higher level requirements normally associated with policies.

Acknowledgements

This work was partially supported by Fundação para a Ciência e Tecnologia through project POSI/32717/00 (FAST—Formal Approach to Software Architecture). We would also like to thank João Pereira for the numerous discussions and comments on the implementation of stock-trading systems.

References

1. L.F.Andrade and J.L.Fiadeiro, “Coordination Technologies for Managing Information System Evolution”, in Proc. CAISE’01, A.Geppert (ed), LNCS, Springer-Verlag 2001, in print.
2. L.F.Andrade and J.L.Fiadeiro, "Interconnecting Objects via Contracts", in UML'99 – Beyond the Standard, R.France and B.Rumpe (eds), LNCS 1723, Springer Verlag 1999, 566-583.
3. L.F.Andrade and J.L.Fiadeiro, “Coordination: the Evolutionary Dimension”, in Technology of Object-Oriented Languages and Systems – TOOLS 38, W.Free (ed), IEEE Computer Society Press 2001, 136-147.
4. L.F.Andrade, J.L.Fiadeiro, J.Gouveia, A.Lopes and M.Wermelinger, "Patterns for Coordination", in COORDINATION'00, G.Catalin-Roman and A.Porto (eds), LNCS 1906, Springer-Verlag 2000, 317-322.

5. M.J.Earl: "An Organizational Approach to IS Strategy-Making", in Information Management: The Organizational Dimension, M. J. Earl (ed.) Oxford University Press, 1998.
6. P.Finger, "Component-Based Frameworks for E-Commerce", Communications of the ACM 43(10), 2000, 61-66.
7. The Financial Times Survey. Stock and Derivatives Exchanges. Friday 31 March 2000.
8. D.Gelernter and N.Carriero, "Coordination Languages and their Significance", Communications ACM 35, 2, pp. 97-107, 1992.
9. J.Gouveia, G.Koutsoukos, L.Andrade and J.L.Fiadeiro, "Tool Support for Coordination-Based Software Evolution", in Technology of Object-Oriented Languages and Systems – TOOLS 38, W.Pree (ed), IEEE Computer Society Press 2001, 184-196.
10. M.Hammer and J.Champey. Reengineering The Corporation: A Manifesto for Business Revolution. Nicholas Brealey Publishing, 1998.
11. S.Katz, "A Superimposition Control Construct for Distributed Systems", in ACM TOPLAS 15, 1993 337-356
12. G.Koutsoukos, J.Gouveia, L.Andrade and J.L.Fiadeiro, "Managing evolution in Telecommunications Systems", submitted, accessible at <http://www.fiadeiro.org/jose/papers>
13. B.Meyer, "Applying Design by Contract", in IEEE Computer, 1992, 40-51.
14. The Oblog Corporation, "The Oblog Specification Language", <http://www.oblog.com/tech/spec.html>
15. Charles Schwab web site, <http://www.schwab-worldwide.com>