

Separation of Core Concerns: Computation, Coordination, and Configuration

M.Wermelinger^{1,4}, J.L.Fiadeiro^{1,3}, L.Andrade^{1,2}, G.Koutsoukos², J.Gouveia²

mw@di.fct.unl.pt, jose@fiadeiro.org, landrade@atxsoftware.com, {gkoutsoukos, jgouveia}@oblog.pt

¹ATX Software SA, Alameda António Sérgio 7, 1C, 2795-023 Linda-a-Velha

²Oblog Software SA, Alameda António Sérgio 7, 1A, 2795-023 Linda-a-Velha

³Dep. Informatics, Fac. Sciences, Univ. Lisbon, Campo Grande, 1749-016 Lisboa

⁴Dep. Informatics, Fac. Sciences and Technology, New Univ. Lisbon, 2829-516 Caparica, Portugal

ABSTRACT

Separating concerns helps developers to get a conceptual grip on large software systems, to reuse parts of the system, and to evolve it. We are interested in separating three generic concerns that are part of any software system: computation, coordination, and configuration. For that purpose we propose a three-layer architecture using two new modeling primitives: coordination contracts to define interactions and coordination contexts to govern system reconfiguration. Each layer is superposed in a transparent way on the layer below, which facilitates the modification of coordination and configuration policies to make the system evolve.

CATEGORIES: B.A., A.B., E.A.

1. MOTIVATION

Taming the ever-growing complexity of software development has been a major goal since the early days of programming. Many concepts and supporting technologies have been put forward, like modules, objects, components, design patterns, aspects, architectural connectors, frameworks, and product-line architectures just to name a few. The rationale is always to provide some help to partition huge software systems into conceptually manageable “chunks” in order to improve design, implementation, and comprehension of such systems. Moreover, such partitioning promotes reuse and facilitates maintenance.

However, it turns out that often the partition, to be effective and bring about the mentioned advantages, requires quite some planning of the possible changes to the system in the future. Otherwise the modifications to be performed to the system will conflict with the dependencies that are implicit in the chosen partitioning, and hence may require substantial changes throughout the “chunks”.

Recently, Aspect-Oriented Programming [16] has been advocated as a non-invasive technique to incrementally weave new concerns

into an existing system. Normally such aspects are limited in scope (e.g., tracing an operation throughout the classes that implement it), and are weaved at compile-time homogeneously into all instances of a given class. If different instances of the same class are to behave differently with respect to a given aspect, the aspect has to explicitly code in which cases it is applicable. Moreover, most AOP approaches are specific to some programming language (notably Java).

We are mostly concerned with higher-level abstractions (together with mappings to different technological platforms) to separate more general and fundamental aspects in order to support system evolution for agile businesses. More precisely, we want to provide modeling primitives to describe coordination and configuration aspects of a system, and to achieve technological support for reconfiguring the interactions among given computational instances in a non-intrusive way at run-time. The rationale for our research focus is as follows.

Most often, the nature of changes that occur in the business are not at the level of the components that model business entities, but at the level of the business rules that regulate the interactions between the entities, and at the level of the business policies that govern the application of business rules. Therefore, we believe that successful methodologies and technologies will have to provide abstractions that reflect the architecture of such systems by supporting a clear separation between computation, as performed by the core business components, coordination, as prescribed by business rules, and configuration, as indicated by business policies.

Moreover, such rules and policies change over time, in some domains (like banking, and telecommunications) quite rapidly to achieve differentiation from the competition [8]. Hence, it is important to provide modeling primitives, with precise semantics and technological support, to allow system designers and developers to easily define and implement new business rules and policies or change existing ones.

Finally, different business rules may apply to different instances of the same business entity class: e.g., the rules that regulate how customers may interact with their accounts may depend on the status of the actual account (say, its balance) and not only on the account type. Furthermore, the same instance may be affected by different rules during its lifetime. Hence, it is of great importance that rules be applied on instances (and not on types) and that they be turned on or off at run-time.

The next section provides more detail about the overall approach, and the remaining two sections summarize our modeling primitives for coordination and configuration.

2. SEPARATION IN THREE LAYERS

The rationale for the methodology and technology that we are building is the strict separation between three aspects of any software system: the computations it performs, the coordination of those computations, and the system configuration.

One of the main reasons for advocating the separation of these concerns is that it facilitates the evolution of systems, because changes that do not require different computational properties can be brought about either by reconfiguring the way components interact, or adding new connectors that regulate the way existing components operate, instead of changing the components themselves. The first two options can be achieved by *superposing* [11] the new coordination and configuration mechanisms on the components, while the latter has side effects on all components that use the services provided by the changed components.

For this approach to evolution to be effective, both for system design and implementation, two requirements must be met. Firstly, the layering must be strict, in the sense that each layer makes use of the services of the layer(s) below. However, a layer must not be even aware there is layer above: if the components have built-in dependencies on the way they will be coordinated and configured, it will be much harder to evolve the system because it is usually very difficult to anticipate which coordination and configuration mechanisms will be needed to respond to change in the application or technological domains.

Achieving this strict separation requires a very strong discipline during domain analysis. The stakeholders have to be quite clear about what is "stable" (i.e., belongs to the core business functionality) and what is "unstable" (i.e., is likely to change in the future), what behavior is purely computational in nature and what can emerge from the configuration of interactions. This is a methodological problem for which there will never be a "solution" in the sense that the quality of the evolutionary model will always depend on the expertise of the stakeholders. Nevertheless, we are building sufficient expertise in using the technology in different application domains [e.g. 12, 13], which will enable us to develop guidelines that will help to steer domain analysis towards the identification of different layers of change.

The separation of coordination from computation has been advocated for a long time in the Coordination Languages [9] community, and the separation of all three concerns is central to Software Architecture, which has put forward the distinction between components, connectors and architectures [5]. The Configurable Distributed Systems community [6], in particular the Configuration Programming approach [15], also gives first-class status to configuration. However, these approaches still do not provide a satisfying way to model the three concerns in a way that supports evolution. Coordination languages do not make the configuration explicit or have a very low-level coordination mechanism (like tuple spaces); architecture description languages do not handle evolution or do it in a deficient way; configuration programming is not at the modeling level.

Our main responsibility in proposing a new model is, therefore, to provide the means through which the proposed separation can be supported, both at the modeling and deployment phases. Hence, the second requirement is that the coordination and configuration aspects must be encapsulated in conceptual units with a precise

semantics and implementation strategy. We have termed those units "coordination contract" [3] and "coordination context", respectively.

3. COORDINATION

In general terms, a coordination contract is a connection that can be established between a group of objects through which rules and constraints are superposed on their joint behavior, thus enforcing specific forms of interaction or adaptation to new requirements. The generic form of a coordination contract is as follows:

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

The classes of objects over which contract instances can be applied are identified under *participants*. A contract may also specify 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. Contracts can be unary, i.e. they can apply to single object instances, e.g., to regulate or adapt the way they behave to new requirements or simply monitor their behavior without inducing any new properties. They can also involve many partners, in which case they behave as synchronization agents that coordinate the way partners interact.

Each coordination rule is of the form:

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

The condition under "when" identifies a trigger. The trigger can be a condition on the state of the participants, a request for a particular service, or an event issued by one or more participants. Several conditions can be placed in the "when" clause. The "do" clause identifies the reactions to be performed, usually in terms of actions of the partners and some of the contract's own actions. When the trigger corresponds to the call for an operation, three types of actions may be superposed on the execution of the operation:

1. **before:** to be performed before the operation
2. **replace:** to be performed instead of the operation (alternative)
3. **after:** 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 is the following: first, all the "before" actions are performed, then one "replace", and finally all the "after" actions. It should be noted that the semantics of contracts allow 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 must adhere to whatever specification clauses apply to the operation (e.g. contracts in the sense of Meyer [14] specifying pre- and post-conditions). This ensures that the functionality of the original operation, as advertised through its specification, is preserved. Furthermore, the set of actions identified under the "do" clause is executed atomically in a

transactional mode. We call this set the “synchronization set” associated with the rule. Hence, the name of the rule acts as a “rendez-vous” in the sense of [7]. Finally, the “with” clause allows us to strengthen the guards that determine the conditions under which the actions in the synchronization set are allowed to occur. If the condition under “with” is false when the trigger is activated, the synchronization set is not executed and the trigger is considered to have failed.

For a detailed description of coordination contracts and their formal semantics, the reader is urged to consult [1,2,3]. In what follows we present an example from banking to motivate the scope and solutions coordination contracts can offer.

Consider a world of bank accounts in which clients can, as usual, make withdrawals. The object class `account` is usually specified with an attribute `balance` and a method `withdrawal` with parameter `amount`. In a typical implementation one can assign the guard `balance >= amount` restricting this method to occur in states in which the amount to be withdrawn can be covered by the balance. However, as explained in [1], assigning this guard to `withdrawal` can be seen as part of the specification of a business requirement and not necessarily of the functionality of a basic business entity like `account`. Indeed, the circumstances under which a withdrawal will be accepted can change from customer to customer and, even for the same customer, from one account to another depending on its type.

Inheritance is not a good way of changing the guard in order to model these different situations. Firstly, inheritance views objects as white boxes in the sense that adaptations like changes to guards are performed on the internal structure of the objects, which from the evolution point of view of is not desirable. Secondly, from the business point of view, the adaptations that make sense may be required on classes other than the ones in which the restrictions were implemented. In our example, this is the case when it is the type of client, and not the type of account, that determines the nature of the guard that applies to withdrawals. The reason the guard will end up applied to `withdrawal`, and the specialization to `Account`, is that, in the traditional clientship mode of interaction, the code is placed on the supplier class.

Therefore, it makes more sense for business requirements of this sort to be modeled explicitly outside the classes that model the basic business entities, because they represent aspects of the domain that are subject to frequent changes (evolution). Our proposal is that guards like the one discussed above should be modeled as *coordination contracts* that can be established between clients and accounts.

```
contract class standard-withdrawal
participants x : Account; y : Customer;
constraints ?owns(x,y)=TRUE;
coordination
sw : when y.calls(x.withdrawal(z))
      with x.Balance() > z;
      replace x.withdrawal(z)
end contract
```

The constraint means that instances of this contract can only be applied to instances of `Customer` that own the corresponding instance of `Account`. The coordination rule superposes the guard that restricts withdrawals to states in which the balance is greater than the requested amount.

Having externalized the “business rule” that determines the conditions under which withdrawals can be made, we can support

its evolution by defining and superposing new contracts. For instance, consider the following contract that allows for relaxing the functionality of `withdrawal` to situations in which an account may be overdrawn.

```
contract class VIP-withdrawal
participants x : Account; y : Customer;
attributes CONST_VIP_BALANCE: Integer;
          Credit : Integer;
constraints ?owns(x,y)=TRUE;
          x.AverageBalance() >= CONST_VIP_BALANCE;
coordination
vw: when y.calls(x.withdrawal(z))
      with x.Balance() + Credit > z;
      replace x.withdrawal(z)
end contract
```

The constraints are now strengthened to restrict the application of the contract to instances of `Customer` that have an average balance in the account that is greater than some fixed amount. For such customers, the guard condition has been weakened to allow for a certain credit limit to be used if the balance of the account is not enough to satisfy the requested amount.

So far, the examples have only imposed further conditions on an operation. It is however also possible to change the operation to be called. As an example, consider a contract that a customer may subscribe to instead of `standard-withdrawal`: whenever the balance is less than the amount, instead of occurring a failure, the whole balance would be withdrawn.

```
contract class limited-withdrawal
participants x : Account; y : Customer;
constraints ?owns(x,y)=TRUE;
coordination
lw: when y.calls(x.withdrawal(z))
      replace x.withdrawal(min(z, x.Balance()))
end contract
```

Besides operation calls, triggers may be changes in state. Consider the following scenario, based on a real financial product offered by a Portuguese bank: whenever the balance of the customer’s checking account goes below some threshold, money is transferred from the savings account; whenever it goes above some upper limit, money is transferred to the savings account to earn better interest.

```
contract class automatic-transfer
participants chk, sav : Account;
attributes low, high, amount: Integer;
constraints ?owns(x,y)=TRUE;
coordination
s2c: when chk.Balance() < low
      do amount := min(sav.Balance(),
                      low - chk.Balance());
      sav.withdrawal(amount);
      chk.deposit(amount);
c2s: when chk.Balance() > high
      do amount := chk.Balance() - high;
      chk.withdrawal(amount);
      sav.deposit(amount);
end contract
```

Such triggers are also useful for warning the user about some exceptional condition. For example, `VIP` customers may be

warned when the balance of their accounts becomes negative, which is allowed but nevertheless not a desirable situation.

```
contract class negative-balance
participants x : Account; y : Customer;
attributes CONST_VIP_BALANCE: Integer;
constraints ?owns(x,y)=TRUE;
x.AverageBalance() >= CONST_VIP_BALANCE;
coordination
nb: when x.Balance() < 0
do write("Negative balance!")
end contract
```

As can be seen above, although coordination contracts are mainly for establishing interactions between components, since they intercept operation calls and state changes, they can also be used for tracing purposes. As another example, consider the following one, to trace any withdrawal on a given account.

```
contract class trace-withdrawals
participants x : Account;
coordination
tw: when *.calls(x.withdrawal(z))
before write("Requested withdrawal:", z);
end contract
```

Notice that the message is output before the actual operation is performed. For example, if the `limited-withdrawal` is also in effect, the actual amount withdrawn may be less than the requested one. Writing **replace** instead of **before** would be an error because the message would appear *instead* of the actual withdrawal and not in *addition* to the withdrawal operation.

It is not necessarily the case that there must always be a reaction to the trigger. Imagine now that an account has to be blocked (i.e., all withdrawals and deposits are disallowed) because its owners are under police investigation. The following unary contract simply falsifies the guard of withdrawals and deposits for any caller. As said earlier, the joint behavior of multiple contracts on the same trigger is "cumulative". Hence, no matter what rules in other contracts are in effect for withdrawals and deposits of the same account, none of them will execute since the conjunction of the guards is false.

```
contract class blocked-account
participants x : Account;
coordination
bw: when *.calls(x.withdrawal(z))
with false;
bd: when *.calls(x.deposit(z))
with false;
end contract
```

As can be seen from these examples, coordination contracts can strengthen conditions and add behavior to interactions among components. Since contracts can be added without changing the components, new business rules can be added on the fly, thus facilitating the evolution of the system.

A more detailed discussion on how contracts can support evolution, as well as a number of examples from different application domains such as banking, telecommunications, and stock trading, can be found in [1,12,13].

We have built an environment that helps programmers to develop Java applications using coordination contracts [10]. More precisely, it allows writing contracts, and to register Java classes (components) for coordination. The code for adapting those components and for implementing the contract semantics is generated based on a micro-architecture that is based on the Proxy and Chain of Responsibility design patterns [4,10]. This micro-architecture handles the superposition of the coordination mechanisms over existing components in a way that is transparent to the component and contract designer. The environment also includes an animation tool, with some reconfiguration capabilities, in which the run-time behavior of contracts and their participants can be observed using sequence diagrams, thus allowing testing of the deployed application¹.

Although contracts make it possible to easily change the business rules that are in effect between the component instances that make up the system, contracts by themselves cannot solve the problems that arise from conflicts or dependencies among the rules. For our example, only one of the `standard-withdrawal`, `VIP-withdrawal` and `limited-withdrawal` can be applied to a customer/account pair, because each contract replaces the `withdraw` operation. Moreover, a `negative-balance` contract can only be applied together with a `VIP-withdrawal`. Such "feature interactions", as they are known in telecommunications, can be made explicit and managed in coordination contexts, as shown in the next section.

4. CONFIGURATION

Components and coordination contracts are the blocks out of which system configuration and evolution is structured. The way computations are carried out locally in components is constrained by the coordination contracts in place, i.e., by the system configuration. On the other hand, the state changes performed by these computations may require the system to be reconfigured. System reconfiguration is achieved through the addition/deletion/substitution of components and/or coordination contracts.

In general, the evolution of the system is not the result of a completely ad-hoc process of reconfiguration. Normally, reconfiguration steps are either programmed or take place in contexts that set constraints on the nature of the operations that can be performed on given configurations. Such contexts capture business activities that can be described in terms of collections of components, coordination contracts that can be superposed on them, and the rules that define the ways in which this superposition can/must take place. For instance, in the banking domain that we have been using as example, coordination contexts are normally defined around customers through which the relationships that customers may hold with their various accounts are managed according to the packages that the bank offers. For instance, such contexts are made available to bank managers each time the customer goes to a branch, or to the customer itself through the Internet or an ATM.

What we will be proposing in this section is a first step towards a modeling primitive that captures a possible pattern of evolution that embodies certain business policies of the organization. We have called this primitive "coordination context" in the sense that it sets up a pattern of evolution with rules that capture constraints

¹ See our demo "Developing and Evolving Java Applications using Coordination Contracts" at OOPSLA'01.

on the way contracts can be superposed on given components when in given states. These constraints add to whatever invariants, etc. have been declared for the contracts and should reflect properties that are specific to a business context.

Each context has a local state that consists of the projection of the global system configuration to the components and coordination contracts declared in the context. This projection defines a “subsystem”. However, the notion of “subsystem” cannot be identified with that of coordination context as a modeling primitive: a context defines a subsystem implicitly but the same or overlapping subsystems may be associated with other contexts. Furthermore, contexts can define subsystems at different levels of abstraction, thus acting as a powerful structuring mechanism for system development.

Contexts should not be treated as “normal” components in the sense that they are not used in configurations to add new functionalities to the system. That is to say, they are not defined in order to contribute to the functional properties that the system can exhibit but only to manage the way the system is allowed to evolve; they model actors as in use cases, i.e. the mechanisms through which “users” (regardless of whether they are human, physical, software, etc) have access to the system, except that, now, such users can interfere with the configuration of the system, not just with its state.

A coordination context may have a state of its own in order to represent information that only makes sense in that business context, e.g., “attributes” of the subsystem identified by the context. For instance, properties like “average balance”, where the average is taken over all the accounts that a given customer owns, require a context in which all these accounts are present.

The syntax that we are developing for this modeling primitive can be illustrated as follows:

```

coordination context customer-space (c:customer)
workspace
component types Account, Customer
contract types
    standard-withdrawal, VIP-withdrawal,
    limited-withdrawal, automatic-transfer,
    negative-balance
constants min-VIP: money
attributes avg-balance
invariants
    exists c, main:account and
    forall a:account (c.owns(a) and
    exists standard-withdrawal(c,a) xor
    exists VIP-withdrawal(c,a) xor
    exists limited-withdrawal(c,a) and
    (exists negative-balance(c,a) implies
    exists VIP-withdrawal(c,a))) and
    (forall a1,a2:account exists
    transfer(a1,a2) implies not exists VIP-
    withdrawal(c,a2))
configuration services
new_account(a:account, L:list_customers, m:money):
pre: not exists a
post: exists a and c.owns(a) and
exists standard-withdrawal(c,a) and
a.balance()==m and (l:L) (l.owns(a) and
exists standard-withdrawal(l,a))
new_transfer(chk, sav:account):
pre: exists chk, sav and
not exists automatic-transfer(chk, sav)
post: exists automatic-transfer(chk, sav)
subscribe_VIP(a:account,V:money):
pre: exists a and c.owns(a) and
not exists VIP-withdrawal(c,a)
post: exists VIP-withdrawal(c,a) and

```

```

VIP-withdrawal(c,a).Credit=V
configuration rules
VIP-to-std:
when exists VIP-withdrawal(c,a)
and avg-balance < min-VIP
post exists standard-withdrawal(c,a)
end context

```

Each instance of a coordination context is “anchored” to a component or set of components. In the example, the anchor is a customer instance, referred to as *c* in the definition of the context (type).

Configuration services correspond to operations for ad-hoc reconfiguration, i.e. on demand from users of the system, whereas configuration rules correspond to different ways of programmed reconfiguration, i.e. to the ability of the system to reconfigure itself in reaction to external events or internal state changes. Notice the use of a post-condition in the configuration rule instead of a specific (trans)action to be performed as a reaction. Together with the use of pre/post-conditions in the definition of services, this allows us to separate context interface from implementation, which adds to flexibility by allowing the choice of the actual reconfiguration operation to depend on “lower level” issues. This separation seems also important for allowing different specializations of the same context to be given different operations according to the “nature” of the access mode that the specialized context is offering (e.g. Counter at local branch, Internet, ATM, etc.).

All operations have to preserve the invariants in addition to the specific pre- and post-conditions. The example’s invariant states that exactly one of the withdrawal contracts is superposed on each customer/account pair. The `VIP-to-std` rule only states a `standard-withdrawal` contract is created for every VIP customer/account pair if the average balance of the customer’s accounts gets below some threshold. Together with the invariant, this implies that the `VIP-withdrawal` contract is removed.

Configuration services and rules are defined over basic configuration operations such as creation and deletion of component and contract instances. Therefore, they subsume primitives typical of object-oriented programming like creation operations. Typically, the programmed configuration rules capture more dynamic properties that require specific actions to be taken in reaction to certain state changes, for instance to restore consistency with respect to policies like the ones that regulate VIP-status for customers.

Indeed, the pre/post-conditions capture business policies that should be elicited during analysis like the dependencies that regulate the subscription of the different account packages. The same applies to the invariants associated with the context: they constrain the creation of the “anchor” of the context, which in the example is a customer instance: when a customer is created, an account *main* needs to be automatically assigned. In the example, the notation should make these aspects sufficiently self-evident, namely the use of the predicate **exists** to indicate whether a given component or contract instance is part of the current configuration, i.e. of the subsystem defined by the context.

5. CONCLUDING REMARKS

As many others, we argue that system development should try to separate as strictly as possible the fundamental concerns of computation, coordination, and configuration, in order to achieve the levels of comprehension, reuse, and evolution that today’s large and ever-changing software systems require.

We propose to use explicit modeling primitives to define coordination and configuration policies in a straightforward and flexible way. The main structuring principle is superposition — coordination contracts are superposed on components, and coordination contexts are superposed on components and contracts — which makes it possible to evolve the system through addition of new contracts and contexts without changing the layer(s) below, and to reconfigure the system at run-time through (un)plugging of instances of contexts, contracts, and components. Moreover, contract and context types are instantiated for specific component (and contract) instances, hence providing great flexibility in defining which coordination rules are in effect for which components, and which configuration policies apply to which parts of the system.

We have already a preliminary implementation of contracts for Java, and our future work will deal with the implementation of contexts, the improvement of the modeling primitives, and the extension to distribution concerns, among other issues.

6. REFERENCES

- [1] L.F.Andrade and J.L.Fiadeiro, "Coordination Technologies for Managing Information System Evolution", in Proc. CAISE'01, K.Dittrich, A.Geppert and M.Norrie (eds), LNCS 2068, Springer-Verlag 2001, 374-387.
- [2] L.F.Andrade and J.L.Fiadeiro, "Coordination: the Evolutionary Dimension", in Technology of Object-Oriented Languages and Systems – TOOLS 38, W.Pree (ed), IEEE Computer Society Press 2001, 136-147.
- [3] 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.
- [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] L. Bass, P. Clements, R. Kazman. *Software Architecture in Practice*, Addison-Wesley, 1998.
- [6] *Proc. of the 4th Intl. Conf. on Configurable Distributed Systems*, IEEE Computer Society Press, 1998.
- [7] N.Francez and I.Forman, *Interacting Processes*, Addison-Wesley 1996.
- [8] Carrol W. Frenzel, *Management of Information Technology*, (3rd ed.), Course Technology, Cambridge, MA, 1999.
- [9] D.Gelernter and N.Carriero, "Coordination Languages and their Significance", *Communications ACM* 35, 2, pp. 97-107, 1992.
- [10] 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.
- [11] S.Katz, "A Superimposition Control Construct for Distributed Systems", *ACM TOPLAS* 15(2), 1993, 337-356.
- [12] G.Koutsoukos, J.Gouveia, L.Andrade and J.L.Fiadeiro, "Managing evolution in Telecommunications Systems", in Proc. IFIP Working Conference on Distributed Applications and Interoperable Systems, Kluwer, in print.
- [13] G.Koutsoukos, T.Kotridis, L.Andrade, J.L.Fiadeiro, J.Gouveia and M.Wermelinger, "Coordination Technologies for Business Strategy Support: a case study in Stock Trading", ECOOP 2001 Workshop on Object Oriented Business Solutions (WOBS01).
- [14] B.Meyer, "Applying Design by Contract", *IEEE Computer*, Oct.1992, 40-51.
- [15] J. Kramer. "Configuration Programming – A Framework for the Development of Distributable Systems", *Proc. CompEuro'90*, pp. 374-384, IEEE, 1990.
- [16] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, J. Irwin. "Aspect-Oriented Programming", in ECOOP'97, LNCS 1241, Springer-Verlag.