# Architecture Based Evolution of Software Systems

Luis Filipe Andrade[1] and José Luiz Fiadeiro[2]

[1]ATX Software S.A., Alameda António Sérgio 7 – 1 C,
2795-023 Linda-a-Velha, Portugal
landrade@atxsoftware.com

[2]Department of Computer Science, University of Leicester
University Road, Leicester LE1 7RH, UK
jose@fiadeiro.org

**Abstract.** Although architectural concepts and techniques have been considered mainly as a means of controlling the complexity of developing software, we argue, and demonstrate, that they can play a vital role in supporting current needs for systems that can evolve and adapt, in run-time, to changes that occur in the application or business domain in which they operate.

## 1 Introduction

Our contribution is motivated by the growing need that companies and organisations have for software systems that can react and adapt, in a flexible and timely way, to changes occurring in the application domain, the technologies that support their deployment, or the more general business infrastructure in which they operate. In other words, whereas the pressure on the Software Engineering community used to be for coming up with methods and techniques that address the complexity of *constructing* software systems, a new challenge is there for controlling the complexity of the *evolution* of such systems. After a brief overview of the context in which these challenges have been arising, we identify the shortcomings of current methods and techniques to address them, and suggest that current work on Software Architectures has a very important contribution to make. Our own contribution, the "three Cs", is then expanded in the rest of the paper.

### 1.1 All Change!

Over the last few years, we have witnessed a significant shift in the way organisations are structured and operate by becoming "information-based". This shift is having tremendous implications in society and the economy in general, with consequent levels of pressure on all those engaged in the software industry. As, more and more, business is based on, or directly operated through, organisations' information systems, people in the software areas are required to work out "miracles" in keeping up with the ever changing expectations of the commercial and strategy management areas.

M. Bernardo and P. Inverardi (Eds.): SFM 2003, LNCS 2804, pp. 148-181, 2003.

Indeed, information systems are now at the core of the competitive edge of every organisation, and all traditional business thrusts such as *differentiation, cost, innovation, growth, alliances* and *time* have a direct impact on the way systems have to be conceived, developed and maintained [18]. Systems must now provide *different* services by making *innovative* use of technological advances (Internet, wireless technology, and so on) and, at the same time, be easily *integrated* with other systems in order to support business *alliances.* In the rules of the "new" or "now"-Economy, this is supposed to happen in "real-time", or "just-in-time", and, as usual, at low cost!

In other words, *agility* is, now, the prime feature required of any information system. As a consequence, software teams are struggling to compete with speeding business and technology evolution, making the ability to *change* systems a more important goal than the ability to build them *ab initio*. Quoting directly from [14], "… 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".

All this means that the "complexity" of software has definitely shifted from *construction* to *evolution*, and that methods and technologies are required that address this new level of complexity and adaptability. We believe that, similarly to what happened with software construction, complexity in evolution is, first of all, a problem of level of abstraction. That is to say, we need to find the abstractions that allow us to expose the "structure" of the problem. Because evolution, as characterised above, is driven by the business domain – be it either as a consequence of changes in the business rules or a merger of enterprises that share some part of the market – it seems to make sense to look for software structures that reflect the structure of the domain itself.

To meet this goal, there are two important aspects that need to be accounted for. On the one hand, we need *modelling* approaches that direct us to the identification of components that relate directly to business entities, and to component interconnections that reflect the business rules according to which the business entities interoperate. On the other hand, we need to provide implementation solutions that are *compositional* with respect to the structures offered by the modelling primitives. By compositionality we mean the ability for the structure obtained at the modelling level to be reflected directly at the implementation level so that changes operated at the level of the business model do not require a global reconfiguration of the system but, rather, only local changes that can be performed at run-time, without interruption of the other services being offered by the system.

## 1.2 No Change with OO

In order to address these new levels of complexity, organisations are looking for answers mainly in the context of object-oriented (OO) development techniques (usually based on the UML [9]), component-based (CB) frameworks [42], design patterns [15] and, most recently, aspect-oriented programming [10]. However, although these techniques have proved useful in taming the complexity of software *construction*, they cannot cope, on their own, with the levels of agility required for *evolution*.

Object-oriented techniques such as inheritance and clientship are too "static" and "white box" when it comes to change. The use of inheritance requires us to know, understand, and modify the internals of objects. In many circumstances, this may not be acceptable (e.g. for economic or security reasons) or even possible (in the case of third-party, closed components, e.g. legacy systems). On the other hand, because interactions in object-oriented approaches are based on *identities* [20], in the sense that, through clientship, objects interact by invoking specific methods of specific objects (instances) to get something specific done, the resulting systems are too rigid to support the identified levels of agility [40]. Any change on the collaborations that an object maintains with other objects needs to be performed at the level of the code that implements that object and, possibly, of the objects with which the new collaborations are established. See [3] for an expanded explanation around an example.

Naturally, object-oriented technology does not prevent more flexible modes of interconnections to be *implemented*. Design mechanisms that make use of event publishing/subscription through brokers and other well-known patterns [15] have already found their way into commercially available products that support implicit invocation [36] instead of feature calling (explicit invocation). However, solutions based on the use of design patterns or, for that matter, Aspect-Oriented Programming, are not at the level of abstraction in which the need for change arises and needs to be managed. Being mechanisms that operate at the design level, there is a wide gap that separates them from the business modelling levels at which change is better perceived and managed. This conceptual gap is not easily bridged, and the process that leads from the business requirements to the identification and instantiation of the relevant design patterns is not easily documented or made otherwise explicit in a way that facilitates changes to be operated. Once instantiated, design patterns code up interactions in ways that, typically, requires evolution to be intrusive because they were not conceived to be evolvable; as a consequence, most of the times, the pattern will dissolve as the system evolves. Therefore, we need semantic primitives through which interconnections can be externalised, modelled explicitly, and evolved directly as representations of business rules.

## 1.3 Architectures for Change

Our own experience in developing applications in one of the most volatile business areas, banking, has shown that the changes that are most often required on a system do not concern the basic computations performed by its components but the way they interact. Indeed, we often forget that the global behaviour of a system emerges both from the local computations that are performed by its components *and* the way they are interconnected. The functionalities that make a system useful cannot be achieved by its components in isolation but only through the collaborations that are established between them. Hence, in very dynamic areas, the most frequent changes are likely to occur not at the level of the core entities of the domain (say, the notion of a bank account) but of the business rules or the protocols that determine how these core entities interact (say, how customers interact with their accounts). Hence, there is an important role to be played by methodological concepts and supporting technology that promote interactions as first-class entities, leading to systems that are "exoskeletal" in the sense that they exhibit their configuration structure explicitly [26].

This is why we believe "Software Architectures" have an important contribution to make: architectural concepts and techniques, as described in [8,38,41], promote a gross decomposition of systems into *components* that perform basic computations and *connectors* that ensure that they interact in ways that make required global system properties to emerge. Such an externalisation of the interconnections should allow for systems to be reconfigured, in a compositional, non-intrusive way, by acting directly on the entities that capture the interactions between components, leading to an evolution process that follows the architecture of the system.

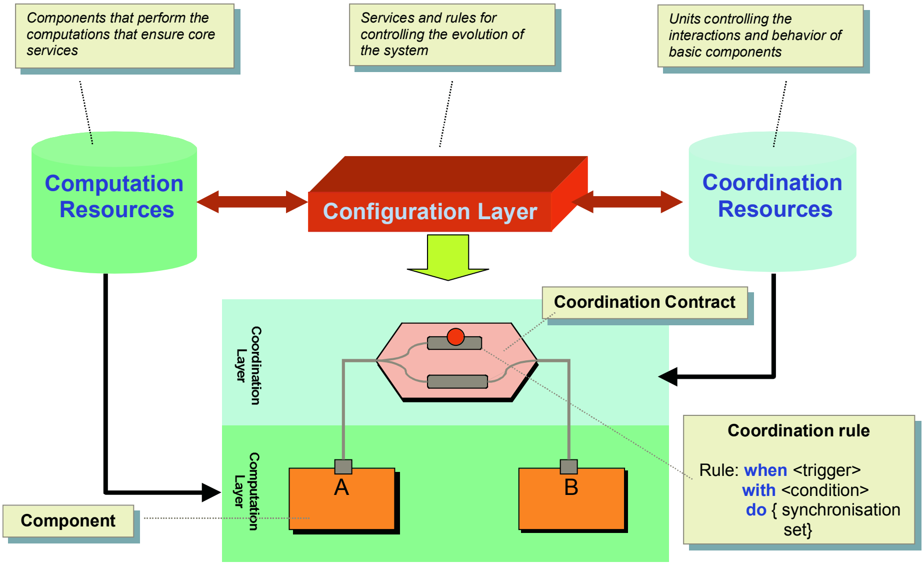
The scope of Software Architectures intersects several other research areas in Software Engineering and Programming Languages. For instance, the crucial separation between "Computation" – the way basic functionalities of system services are ensured – and "Coordination" – the mechanisms through which components interact – has been the core subject of the community that gathers around "Coordination Languages and Models" [16]. Mechanisms through which systems can be dynamically reconfigured have been developed in the area of "Configurable Distributed Systems" [25]. In this paper, we provide an overview of an approach to system evolution that brings together all these different streams of research and has been tested in several application domains [4,23,24]. In section 2, we further motivate and outline the basic principles of the approach. In section 3, we address the way business rules can be modelled through semantic primitives that, basically, amount to architectural connectors. In section 4, we focus on supporting evolution through programmed and reactive dynamic reconfiguration. In section 5, we address the way the approach can be automated and supported by tools. Finally, in section 6, we conclude with comparisons with other work and an outlook of current and planned developments.

## 2 The Three Cs of Architectures

The architectural approach that we will be describing is based on the crucial separation between "three Cs": Computation, Coordination, and Configuration. This separation needs to be supported at two different levels. On the one hand, through semantic primitives that address the "business architecture", i.e. the means that need to be provided for modelling business entities (Computation), the business rules that determine how the entities can interact (Coordination), and the business contexts through which specific rules can be superposed, at run-time, to specific entities (Configuration). On the other hand, the architectural properties of the deployment infrastructures that can carry this distinction to the design and implementation layers, and support the levels of agility identified in the previous section. We start by discussing the latter. An overview of the semantic primitives is then given as a motivation for the subsequent sections, which explain them in detail.

## 2.1 The CCC System Architecture

As already mentioned, the rationale for the methodology and technologies that we have been building is in the *strict* separation between three aspects of the development and deployment of any software system: the *computations* performed locally in its components, the *coordination* mechanisms through which global properties can emerge from those computations, and the *configuration* operations that ensure that the system will evolve according to given constraints such as organisational policies, legislation, and other. Such layering should be strict in order to allow for changes to be performed at each layer without interfering with the levels below.


Components that perform the computations that ensure core services
Services and rules for controlling the evolution of the system
Units controlling the interactions and behavior of basic components
Computation Resources
Configuration Layer
Coordination Resources
Coordination Contract
Coordination Layer
Computation Layer
A
B
Coordination rule
Rule: when <trigger>
with <condition>
do { synchronisation set}
Component

The Computation Layer should contain the components that perform the computations that ensure the basic services provided within the system. Each component has two interfaces: a functional interface that includes the operations that allow one to query and change the encapsulated component state; and a configuration interface that provides the component's constructors and destructors, and any other operations that are necessary to the correct management of dynamic reconfiguration. One such operation is querying whether the component is in a "stable" state in which the component may be deleted or its "connections" to other components can be changed; another example is an operation that can temporarily block the component's execution while a reconfiguration that involves it is processed. The reason for separate interfaces is to be able to constrain the access that the various parts of the architecture have to each other and, hence, achieve a cleaner separation of concerns. In the case of the coordination layer, we require that no component should be able to invoke another component's configuration operations: components should not create other components because that is a change to the currently existing configuration and, as such, should be explicitly managed by the configuration layer.

The Coordination Layer defines the way computational components are interconnected for the system to behave, as a whole, according to set requirements. In the terminology of Software Architecture, this layer is populated by the *connectors* that regulate the interactions between the components of the layer below. We call such connectors *coordination contracts* or, for simplicity, *contracts*. We also require each contract to provide a functional and a configuration interface; each constructor in the configuration interface must include as arguments the components that the connector instance to be created will coordinate. We impose two restrictions: a contract may not use the configuration interface of any contract or component; and a contract may not use another contract's functional interface. The rationale for the first condition is again that configuration operations should only be performed by the configuration layer. The reason for the second condition is to make it possible to evolve the system through (un)plugging of individual contracts between components, which is only possible if there are no dependencies among contracts. The coordination effects that contracts put in place are described in terms of trigger-reaction rules as discussed in section 3.

At each state, the interconnections put in place among the population of basic components via contracts define the current configuration of the system. The Configuration Layer is responsible for managing the current configuration, i.e., for determining, at each state, which components need to be active and what interconnections need to be in place among which components. This layer provides a set of high-level reconfiguration operations that enforce global invariants over the system’s configuration. The actual implementation of the configuration layer may follow the technical architecture given in [34]: a configuration database containing updated information about the current configuration, a consistency manager that enforces a “stable” state in which reconfiguration can occur, and a reconfiguration manager that executes the reconfiguration operations, using the services of the database and the consistency manager. The implementation of the reconfiguration operations makes use not only of the configuration interfaces provided by the components and contracts, but also of the functional interfaces because some changes to the configuration may depend on the current state of components and contracts, and may trigger state modifications to restore application-wide consistency.

Systems whose design architecture supports this separation of concerns through a strict layering can be evolved in a compositional way. 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 performing changes in the components themselves. This can be achieved by superposing, dynamically, new coordination and configuration mechanisms on the components that capture the basic business entities. If the interactions were coded in the components themselves, such changes, if at all possible thanks to the availability of the source code, besides requiring the corresponding objects to be reprogrammed, with possible implications on the class hierarchy, would probably have side effects on all the other objects that use their services, and so on, triggering a whole cascade of changes that would be difficult to control.

On the other hand, the need for an explicit configuration layer, with its own primitives and methodology, is justified by the need to control the evolution of the configuration of the system according to the business policies of the organisation or, more generally, to reflect constraints on the configurations that are admissible (configuration invariants). This layer is also responsible for the degree of self-adaptation that the system can exhibit. Reconfiguration operations should be able to be programmed at this level that enable the system to react to changes perceived in its environment by putting in place new components or new contracts. In this way, the system should be able to adapt itself to take profit of new operating conditions, or reconfigure itself to take corrective action, and so on.

According to the nature of the platform in which the system is running, this strict layering may be more or less directly enforced. For instance, we have already argued that traditional object-oriented and component-based development infrastructures do not support this layering from first-principles, which motivates the need for new semantic modelling primitives as discussed below. However, this does not mean that they cannot accommodate such an architecture: design techniques such as reflection or aspect-oriented programming, or the use of design patterns, can be employed to provide the support that is necessary from the middleware. In fact, we have shown how the separation between computation and coordination can be enforced in Java through the use of well known design patterns, leading to what we called the "Coordination Development Environment" or CDE [5,17,27], which we will discuss in section 5.

The design patterns that we use in the CDE provide what we can call a "micro-architecture" that enforces the externalisation of interactions, thus separating coordination from computation. It does so at the cost of introducing an additional layer of adaptation that intercepts direct communication through feature calling (clientship) and, basically, enforces an event-based approach. In this respect, platforms that rely on event-based or publish-subscribe interaction represent a real advantage over object-based ones: they support directly the modelling primitives that we will be discussing in the paper.

## 2.2 The CCC Business Architecture

The separation of coordination from computation has been advocated for a long time in the Coordination Languages community [16], and the separation of all three concerns is central to Software Architecture, which has put forward the distinction between components, connectors and architectures [38]. The Configurable Distributed Systems community [29], in particular the Configuration Programming approach [25], also gives first-class status to configuration. However, these approaches 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 (e.g. tuple spaces); architecture description languages do not handle evolution from first principles or do it in a deficient way; configuration programming is not at the business modelling level.

For instance, the reconfiguration operations that we provide through *coordination contexts* correspond more to what in other works is called a reconfiguration script [11] than the basic commands provided by some ADLs to create and remove components, connectors, and bindings between them [30]. Coordination contexts also make explicit which invariants the configuration has to keep during evolution. It is natural to express these invariants in a declarative language with primitive predicates to query the current configuration (e.g., whether a contract of a given type connects some given components). Such languages have been proposed in Distributed Systems (e.g., Gerel-SL [11]) and Software Architecture approaches (e.g., Armani [35]). However, all these approaches *program* the reconfiguration operations, i.e. they provide an operational specification of the changes. Our position, as expanded in section 4, is that, at the modelling level, those operations should also be specified in a declarative way, using the same language as for invariants, by stating properties of the configuration before and after the change. In other words, the semantics of each reconfiguration operation provided in this layer is given by its pre- and post-conditions.

On the other hand, it is true that modelling languages like the UML [9] already provide techniques that come close to our intended level of abstraction. For instance, "use cases" come close to coordination contexts: they describe the possible ways in which the system can be given access and used. However, they do not end up being explicitly represented in the (application) architecture: they are just a means of identifying classes and collaborations. More precisely, they are not captured through formal entities through which run-time configuration management can be explicitly supported. The same applies to the externalisation of interactions. Although the advantage of making relationships first-class citizens in conceptual modelling has been recognised by many authors (e.g. [22]), which led to the ISO General Relationship Model (ISO/IEC 10165-7), things are not as clean when it comes to supporting a strict separation of concerns.

For instance, one could argue that mechanisms like association classes provide a way of making explicit how objects interact, but the typical implementation of associations through attributes is still “identity”-based and does not really externalise the interaction: it remains coded in the objects that participate in the association. The best way of implementing an interaction through an association class would seem to be for a new operation to be declared for the association that can act as a mediator, putting in place a form of implicit invocation [36]. However, on the one hand, the fact that a mediator is used for coordinating the interaction between two given objects does not prevent direct relationships from being established that may side step it and violate the business rule that the association is meant to capture. On the other hand, the solution is still intrusive in the sense that the calls to the mediator must be explicitly programmed in the implementation of the classes involved in the association.

Moreover, the use of mediators is not incremental in the sense that the addition of new business rules cannot be achieved by simply introducing new association classes and mediators. The other classes in the system need to be made aware that new association classes have become available so that the right mediators are used for establishing the required interactions. That is, the burden of deciding which mediator to interact with is put again on the side of clients. Moreover, different rules may interact with each other thus requiring an additional level of coordination among the mediators themselves to be programmed. This leads to models that are not as abstract as they ought to be due to the need to make explicit (even program) the relationships that may exist between the original classes and the mediators, and among the different mediators themselves. In summary, we end up facing the problems that, in the introduction, we identified for the use of design patterns in general.

The primitive – *coordination law* – that we have developed for modelling this kind of contractual relationship between components circumvents these problems by abandoning the "identity"-based mechanism on which the object-oriented paradigm relies for interactions, and adopting instead a mechanism of superposition that allows for collaborations to be modelled outside the components as connectors (coordination contracts) that can be applied, at run-time, to coordinate their behaviour. From a methodological point of view, this alternative approach encourages developers to identify dependencies between components in terms of *services* rather than identities. From the implementation point of view, superposition of coordination contracts has the advantage of being non-intrusive on the implementation of the components. That is, it does not require the code that implements the components to be changed or adapted, precisely because there is no information on the interactions that is coded inside the components. As a result, systems can evolve through the addition, deletion or substitution of coordination contracts without requiring any change in the way the core entities have been deployed.

This is the approach that we are going to present in the rest of the paper. We start by presenting the primitives that support the separation between computation and coordination and the modelling of business rules as architectural connectors – coordination laws and interfaces. We then discuss the primitives – coordination contexts – that we provide for configuring and controlling the evolutionary process.

## 3 Modelling Interactions

The semantic primitive that we provide for externalising interactions – coordination contract – makes available the expressive power of a *connector* in the terminology of software architectures. It consists of a prescription of certain coordination effects (the *glue* of the connector in the sense of [1]) that can be superposed on a collection of partners (system components) when the occurrence of one of the contract *triggers* is detected in the system. Contracts establish interactions at the instance level when superposed on a running system as the result of the instantiation of a *coordination law*. These capture connector types as in Architecture Description Languages. In the description of a coordination law, the nature of the partners over which the law can be instantiated are identified as *coordination interfaces* (the *roles* of the connector type in the sense of [1]). These act as types that can be instantiated with components of the system when a contract that instantiates the law needs to be activated on a particular configuration of the running system.

### 3.1 Laws and Interfaces for Coordination

We are going to illustrate our approach with a very simple example from banking. As a core concept of banking, we assume that an account offers three basic services – `balance():money`, `credit(n:money)`, and `debit(n:money)`. The methodology that we are building around the CCC-approach suggests that, in order to model specific business activities that involve bank accounts, we should superpose whatever coordination mechanisms are required to model business rules as external entities, rather than specialise the class with new, "hard-wired" features each time a new business rule, or changes to existing business rules, come into play.

For instance, restrictions on debits should not be "hard-wired" as pre-conditions on the basic method that performs the debit but, rather, as contracts that coordinate specific interactions that involve the method. As a business activity, a withdrawal by a given customer is an operation that involves both an account and a customer, regardless of the way that the customer is represented in the system, i.e. of whether it is a software component in the information system, an interface object that captures an external agent of the system, etc. Hence, we model customers through a coordination interface that is agnostic with respect to any specific form of representation and allows for such representations to evolve as new decisions are made during the lifetime of the system. In terms of our example, we assume that customer interfaces include `owns(a:account):Boolean`, and `withdrawal(n:money,a:account)`.

```
coordination interface customer-withdrawal
import types  money, account;
services      owns(a:account):Boolean
events        withdrawal(n:money; a:account)
end interface
```

The difference between *services* and *events* is quite simple: *services* identify operations that instances of the interface need to provide for the contract to operate according to the law; *events* identify situations produced during system execution that are required to be detected as triggers for the contract to react and activate a *coordination rule* as discussed below. For the proposed coordination mechanism, we are required to detect as triggers events that consist of customers performing withdrawals, and be provided with services that query about the account ownership relation.

In the traditional, object-oriented way, typical events are feature calls. In fact, in an object-oriented setting, a withdrawal would normally be modelled as a direct call to the debit operation of the corresponding account. That is to say, as part of the design of the system, `withdrawal(n,a)` establishes a call `a.debit(n)`. As argued in [3,40], this form of direct invocation leads to systems in which components are too tightly coupled, making any change on the interactions to be intrusive on the code that implements the components. The alternative that we propose through coordination contracts consists in providing an explicit representation of the interaction outside the components that can be superposed on them when needed without their knowledge.

The coordination interface that corresponds to the other partner of this business rule is as follows:

```
coordination interface account-debit
import types money;
```

```
services     balance():money;
             debit(a:money)  post balance() = old balance()-a
end interface
```

The inclusion of properties in an interface, e.g. the pre and post-conditions on debits, is meant to provide means for requirements to be specified on the components that can be bound to the interface and become under the coordination of the law. In this example, we are stating minimal properties on the functional behaviour of the services included in the interface, namely that debits interact with observations of the balance as expected.

Given these two coordination interfaces, we can specify the law that models standard withdrawals as follows:

```
coordination law standard-withdrawal
partners a:account-debit; c:customer-withdrawal
rules        when c.withdrawal(n,a)
             with a.balance() • n & c.owns(a)
             do   a.debit(n);
end law
```

Besides identifying the coordination interfaces, a coordination law specifies the rules that apply to the partners over which it is instantiated in order to coordinate their interaction. Such coordination rules are of the form:

```
when condition
with condition
do   set of operations
```

Each coordination rule identifies, under the "when" clause, a trigger to which the contracts that instantiate the law will react – a request by the customer for a withdrawal in the case at hand. The trigger can be just an event observed directly over one of the partners or a more complex condition built from one or more events. Under the "with" clause, we include conditions (guards) that should be observed for the reaction to be performed. If any of the conditions fails, the reaction is not performed and the occurrence of the trigger fails. Failure is handled through whatever mechanisms are provided by the language used for deployment.

The reaction to be performed to occurrences of the trigger is identified under the "do" clause as a set of operations – a debit for the amount and on the account identified in the trigger. This set may include services provided by one or more of the partners as well as operations that are proper to the law itself (as illustrated in section 3.2). The whole interaction is handled as a single transaction, i.e. it consists of an atomic event in the sense that the trigger reports a success only if all the operations identified in the reaction execute successfully and the conditions identified under the "with" clause are satisfied. Therefore, every coordination rule specified in a law identifies a point of "rendez-vous" in which the partners are brought together to synchronise their lives.

As we have just mentioned, in execution terms, the rendez-vous is an indivisible, atomic action. This paradigm of "joint actions" is present in a number of approaches to parallel program design (e.g. [6]), as well as in recent languages for information system modelling like MERODE [39]. When instantiated over a running configuration, each partner participates in the rendez-vous according to the services declared in the corresponding interface, but unaware of the type of coordination to which it is being subjected. Notice that the execution of the services is performed locally by the components that hold them and, as such, may be subject to further local constraints. Hence, the whole reaction may fail even if the “with”-clause is satisfied because some of the operations involved may not execute successfully.

The “with”-clause plays a fundamental role in the externalisation of business rules in the sense that it allows for the effects of operations (the computations that they perform) to be handled locally, and the conditions under which they are performed to be controlled, totally or partially, at the level of coordination contracts. Deciding what is part of an entity and what pertains to a business rule is not an easy matter and requires a good expertise on the business domain itself. For instance, market evolution has shown that the circumstances under which a withdrawal can be accepted keeps changing as competition dictates banks to come up with new ways for customers to interact with their accounts. Therefore, it should not be too difficult to come to the conclusion that the precondition on debits derives more from the specification of a business requirement than an intrinsic constraint on the functionality of a basic business entity like `account`. Hence, it seems best to shift the precondition to the law so that the contract that regulates how a given costumer interacts with a given account can be replaced without interfering with the code that implements the debit. For instance, a customer may get an upgraded service by subscribing to a VIP-package:

```
coordination law VIP-withdrawal
partners     a:account-debit; c:customer-withdrawal
operations   credit():money
rules        when c.withdrawal(n,a)
             with a.balance()+credit()• n & c.owns(a)
             do   a.debit(n);
end law
```

As illustrated in this example, it is possible to declare features that are local to the law itself. For instance, in the case of VIP-withdrawals, it makes sense to assign the credit-limit that is negotiated between the customer and the bank to the law itself rather than the customer or the account. This is because we may want to be able to assign different credit limits to the same customer but for different accounts, or for the same account but for different owners.

One could argue for a separate partner of the law to be defined for `credit` but, being a feature that is local to the law, it should not be externalised. Indeed, although every contract (instance of the law) has an associated component for implementing these local features, this component should not be public. For instance, a law does not define a public class and its instances are not considered as ordinary components of the system.

This is one of the reasons why association classes, as available in the UML, are not expressive enough to model the coordination mechanisms of contracts. Although coordination laws allow for interactions to be made explicit in conceptual models, they should not be accessed in the same way as the classes that model the core business entities. Contracts do not provide services: they coordinate the services made available by the core entities. Another shortcoming of association classes, and the use of mediators as discussed in the previous section, is that they do not enforce the synchronisation and atomicity requirements of the proposed coordination mechanisms.

We need to stress the fact that coordination interfaces are defined so as to state *requirements* placed by laws on the entities that can be subjected to its rules and not as a declaration of features or properties that entities offer to be coordinated. This means that coordination interfaces should restrict themselves to what is essential for the definition of given laws and hence, in the extreme, can be local to the laws themselves. However, for the sake of reusability and simplification of the binding process, it is useful to externalise coordination interfaces from the laws in the context of which they are defined, and establish a hierarchy between them that is consistent with the compliance relationship in the sense that a component that complies with a given interface also complies with any ancestor of that interface or, that any binder of the component for that interface will also serve as a binder for any ancestor. Hence, in a sense, coordination interfaces fulfil the role of representations of abstract business entities in the sense that the hierarchy of interfaces will, ultimately, provide a taxonomy of all the business uses that are made of entities in the application domain.

Given this, we insist that, as a methodological principle, the definition of coordination interfaces should be driven by the modelling of the business rules as coordination laws and not by the modelling of the entities of the business domain as it is usual in object-oriented and other traditional "product"-oriented approaches. In this sense, it makes no sense to define a coordination interface for accounts in general but, instead, and in the extreme, as many interfaces as the business rules that apply to accounts require (something that is evolutionary in nature because it is as impossible to predict how an information system will evolve as for how the business of an organisation will grow). Ultimately, these will identify all the usages that the specific business makes of the notion of account in a "service-oriented" perspective. As the business rules evolve, new coordination interfaces are defined and placed in the hierarchy. In a product-oriented notion of interface, each such change in the business rules would require a change in the account interface, which is against the spirit of "agility-enhancer" that our method is supposed to deliver.

We should also stress the fact that a binding/compliance mechanism needs to be established for each target development environment. Each time a technological change is introduced that makes it necessary for an application to be redeployed, the binding mechanism will be likely to have to change accordingly and, hence, the compliance relations will need to be re-established or the corresponding binders redefined. Again, some degree of automatic synthesis would be welcome but, at least, computational support should be provided for the redefinition of the binders.

### 3.2 Business Rules as Coordination Laws

The discussion so far has focused on the binding of coordination interfaces to given components of the system that we want to interconnect in order for some global property of the system to emerge from their collaboration. However, coordination interfaces can also act as useful abstractions for either events or services that lie outside the system, or global phenomena that cannot be necessarily localised in specific components. In the case of events, this allows for the definition of reactions that the system should be able to perform to triggers that are either global (e.g. a deadline) or are detected outside the system. In the case of reactions, this allows us to identify services that should be procured externally. This is particularly useful for B2B operations and the modelling of Web-services, the paradigms into which e-business is evolving. What is important is that a uniform notation and semantics is maintained for all cases in order not to compromise changes in the boundaries of the system.

For instance, one could define a law for handling transfers that arrive at a bank from outside by relying on the following interfaces:

```
coordination interface account-credit
import types  money;
services      balance():money;
              credit(a:money)  post balance() =
                          old balance()+a
end interface

coordination interface external-transfer
import types  money, account, transfer-id;
events        transfer(n:money;a:account;t:transfer-id)
end interface
```

When defining the corresponding law

```
coordination law external-transfer-handler
partners    a:account-credit; external-transfer
operations  ackn(t:transfer-id)
rules       when transfer(n,a,t)
            with a.exists
            do   n•1000:a.credit(n-100)
                   & n<1000:a.credit(90%n)
                   & ackn(t)
end law
```

we do not provide any formal parameter of type external-transfer because it is not intended to be bound to any specific component of the system. Notice that the synchronisation set is now more complex than in the previous examples: an acknowledgment is sent and a commission is charged – 100 units if the transfer is above 1000 or a 10% commission for amounts below 1000.

The examples that we have just seen illustrate situations in which a coordination law is defined to model the interaction that is required between certain components of the business domain to ensure some functionality. There are other situations in which coordination laws are defined in order just to *monitor* the behaviour of given components. For instance, assume that new legislation is passed that requires credits over a certain amount to be reported to the central bank – e.g. as a means of detecting money laundering. Rather than revise the implementation of credits to take care of this new requirement, it is better to superpose a contract over every account to perform the required monitoring activity:

```
coordination interface account-credit-event
import types  money;
events        credit(a:money)
end interface
```

```
coordination law report-big-credits
partners     a:account-credit-event
operations   big():money;
             report(n:money);
             set-big(n:money) post big()=n
rules        when  a.credit(n) & n•big()
             do  report(n);
end law
```

Notice that the interface `account-credit` that we defined for the laws on withdrawals did not include debits as events but as services. Hence, we cannot reuse it for this law.

Contracts can also be used for superposing *regulators* over certain components of the system. As an example, consider the situation in which the bank decides to penalise customers who fail to keep a given minimum average balance by charging a monthly commission. This situation can be modelled through the superposition of a contract over every bank account through the following interface:

```
coordination interface average-balance
import types money
services     average():money;
             debit(n:money)
end interface
```

The corresponding law is:

```
coordination law commission-on-low-balance
partners     a:average-balance
operations   minimum():money; charge():money
rules        when  end-of-month
             do  minimum>a.average():a.debit(charge())
end law
```

We are assuming that `end-of-month` is a global trigger that the system detects through some calendar mechanism that we take for granted. Otherwise, we could define a global interface as illustrated for the external transfers.

We are now going to illustrate a more complex use of coordination laws. It refers to a banking product that involves two accounts, typically a checking account and a savings account, transferring money from one to the other in order to keep the balance of one of them – the checking account – between a given minimum and a given maximum. On subscription of this package, the contract is superposed over the two accounts through the following interface:

```
coordination interface account-debit&credit
type-id      account
events       balance():Money;
services     debit(a:Money);
             credit(a:Money);
             balance():Money;

properties   balance() after credit(a) is balance()+a
             balance() after debit(a) is balance()-a
end interface
```

Notice that balance is now included as an event as well. This means that components that are partner instances are required to make available state changes on the current balance as observable triggers.

The proposed law is:

```
coordination law flexible-package
partners     c,s:account-debit&credit
operations   minimum,maximum:money
rules        when c.balance()<minimum
             do   s.debit(min(s.balance(),maximum-c.balance())
                    & c.credit(min(s.balance(),maximum-c.balance())
             when c.balance()>maximum
             do   c.debit(c.balance()-maximum)
                    & s.credit(c.balance()-maximum)
end law
```

The "do" clauses in the coordination rules are now more complex in the sense that the reactions to be performed on occurrence of the triggers involves more than one operation: they both define sets of actions that we call the *synchronisation sets* associated with the rules. As already mentioned, the reactions are executed as transactions, which means that the synchronisation sets are executed atomically. In what concerns the language in which the reactions are defined, the primitives supported by the transaction manager of the target implementation environment should be allowed. However, at higher levels of modelling, we normally use a more abstract notation for defining the synchronisation set as above.

## 4 Configuring and Evolving Interactions

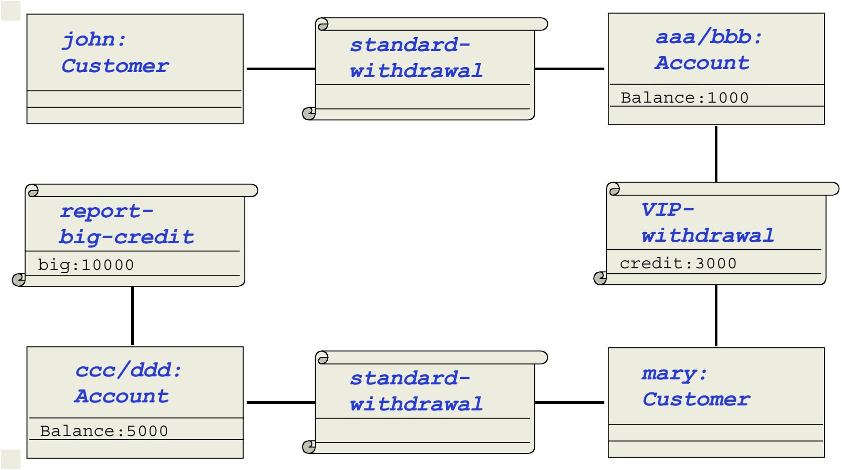
The purpose of coordination laws, as discussed in the previous section, is to provide a level of representation for business rules that is, basically, application-independent. This is because we want to make available a level of business modelling that is invariant with respect to evolution taking place in the “technological axis” and that may lead to changes on the way applications are deployed. It is clear that changes at this level should not induce changes on the business model but only on the way the model is being reified in terms of the specific applications that are being (re)deployed.

In this section, we are concerned with the mechanisms through which laws can be instantiated over running systems and the primitives through which the evolution of the configuration of the system can be controlled so as to ensure global policies of an organisation.

### 4.1 Configuring Interactions

Coordination laws model the collaborations that need to be put in place among components to ensure that the global properties that are required of the system in any given state can emerge from the interactions and the computations that are being performed locally within the components that are present in the configuration of the system, at that same state. In broad terms, a configuration of a system consists of a collection of components that deploy core entities of the business domain and a collection of instances of laws that coordinate the interactions between them. Such instances – called (coordination) *contracts* – are components that execute the coordination rules of the law by reacting to the triggers, evaluating the guards and executing the synchronisation sets.

An example of a configuration in the banking domain that we have been discussing is given below in terms of a diagrammatic notation that should be self-explaining (rectangles represent components and scrolls represent contracts): two customers, mary and john, are joint owners of account aaa/bbb; mary uses a VIP-withdrawal package with credit 3000 and john a standard-withdrawal package; mary is the owner of a second account ccc/ddd with a standard-withdrawal contract and is monitored for credits above 10000.



For the reader who is familiar with our previous publications on "coordination contracts", coordination laws may appear to be a mere syntactic variation. A more careful analysis will show that there are fundamental generalisations whose aim is to provide mechanisms for addressing a more abstract level of "shearing" (to use an "architectural" metaphor). In previous publications, starting with [2], we defined coordination contracts having in mind their use in the context of object-oriented modelling, e.g. through the UML. Hence, in the description of a contract (type), the partners to which contract instances can be applied were identified as object classes of the system and additional rules on the population of these classes. Typically, such classes will be already present or will end up in the class diagram defined for the application.

The need for application-independent level of business modelling led us to generalise this characterisation of the business entities (subjects) to which the business rule applies. This generalisation was made, on the one hand, to remove the bias towards object-oriented approaches and, on the other hand, the direct connection to the class diagram of the application. For this purpose, in coordination laws, subjects are identified in terms of a number of *coordination interfaces* – an abstraction of the notion of *role* of architectural connectors similar to the one formalised in [1]. The use of these laws in a specific application requires *interface instantiation mechanisms* through which one can determine, for any component of the application and interface of a coordination law, whether the component *complies* with the interface in the sense that it can be used as a subject or, better still, if and how it can be adapted to comply and become a subject of the law through that interface. Ideally, this process of instantiation should be automated, leading to the synthesis of the necessary adaptor. A very simple example, but probably the case that is likely to be most frequent in the immediate future use of the concept, is the one in which coordination interfaces correspond to notions like Java-class interfaces and the instantiation mechanism is the one that Java provides through the "implements" relationship between interfaces and classes.

The degree of dynamic reconfigurability that can be achieved through coordination interfaces and laws depends directly on the ability of the execution environment to make available, as recognisable triggers, interactions between entities as well as events taking place within individual entities. For instance, distributed systems and object-based environments are paradigmatic examples in which collections of triggers are explicitly made available. However, and ultimately, a good design architecture (e.g. publish/subscriber) is a means of enabling collections of triggers to be made available in environments that do not support directly any notion of interaction. Compositionality means that triggers can be mapped to meaningful abstractions of the application domain.

The nature of events and services can vary depending on the nature of the language, or class of languages, in which the entities to which the law applies are, or are likely to be, implemented. In an object-oriented environment, typical events that constitute triggers are calls for operations/methods of instance objects, and typical services are the operations that are made public by the object class. In such cases, as already mentioned, coordination interfaces can be identified with abstractions made available in object-oriented programming through mechanisms such as class interfaces in Java, pure virtual classes in C++, and abstract classes in Eiffel. Another class of events that we have found useful as triggers is the observation of changes taking place in the system. For such changes to be detected, components must make available methods through which the required observations can be made (something that the mechanism of interface instantiation must check), and a detection mechanism must be made available in the implementation platform to enable such changes to be effectively monitored (something that is not universally provided).

What is important to stress is that, throughout the paper, we do not address object-based development specifically. Instead, we have aimed for more general notions to which the coordination technologies that we are developing apply. Hence, the notion of component that we assume is the more general one that has been popularised in [42] by which we mean "a unit of composition with contractually specified interfaces and explicit context dependencies only".

A coordination interface is, therefore, something more generic than an object instance type. It represents the signature of services and events that a component (software module) has to offer to be "compliant" with that interface, i.e. for that interface to be able to be instantiated by that component. This means that we cannot assume that any given interface is going to be instantiated necessarily, or always, by an object class. In the cases in which this does happen, the interface can be seen as an abbreviation of the use of the type associated with that class. These remarks should not be taken lightly because they can, and should, have a profound impact in the methods and languages that we use for developing systems. For instance, by assuming that we are not dealing necessarily with object instances implies that we cannot use instance operations during the specification of interfaces, something far too radically distant from for what any object-oriented language can offer.

When a coordination interface is instantiated with a specific component as part of the activation of a law over a given configuration of the system, the features declared in that interface must be formally related with what the component makes available through its own interface. Again, this process of instantiation will depend on the nature of the deployment of the component itself. For instance, as already mentioned, programming languages such as Java provide mechanisms for interfaces to be implemented through object classes. Ideally, the component definition language should support the distinction between *event* (to be used for triggers) and *services* (to be used for reactions), but this is not necessary. We decided to separate concerns in coordination interfaces as much as possible as a means of setting the direction in which we think Interface Definition Languages could evolve, but this separation does not need to be enforced for our techniques to be applicable. Hence, for instance, typical notions of interface in which components declare which methods are public can be used: events can be detected as calls to the public methods of the component and services can be effected through the invocation of these methods by the contract.

This separation between the coordination interface and the components themselves is, indeed, an essential mechanism for being able to interconnect heterogeneous components and, hence, not to compromise the ability of the system to evolve by upgrading its basic components or integrating third-party components. Furthermore, it enforces the required degree of "business centricity" in the sense that a coordination law, as a direct representation of a business rule, should apply, through contracts that enforce that law, to whatever components embody the business entities involved in the rule. In other words, the coordination interfaces that constitute the subject roles of the law act as abstract descriptions of the business entities that can become involved in instantiations of any contract that enforces the law.

### 4.2 Evolving Interactions

The use of coordination contracts for representing business rules leads to an approach to the evolution process that is based on reconfiguration techniques as known from Configurable Distributed Systems [25,29]. At each moment of the life of the system, its configuration can be identified with the collection of components that have been created (but not destroyed) interconnected through the contracts that will have been superposed among them. As part of the evolution process, new components may be added, existing components may be removed, new contracts may be superposed, existing contracts can be removed or replaced, and so on. All these operations rewrite the configuration, producing a new configuration for the system. The new configuration will dictate how the system will behave from then on, computationally speaking, through the revised set of components and interactions among them. On the other hand, as a result of the computations performed by the components, or the interactions that are maintained with the environment, the global state of the system changes, which may require a reconfiguration to adapt it to the new circumstances.

Having mechanisms for evolving systems is not the same as prescribing when and how these mechanisms should be applied. Evolution is a process that needs to be subject to rules that aim at enforcing given policies of organisations over the way they wish or are required, e.g. through legislation, to see their businesses conducted. For this purpose, we provide a modelling primitive – coordination contexts – through which the reconfiguration capabilities of the system can be automated, both in terms of ad-hoc services that can be invoked by authorised users, and programmed reconfigurations that allow systems to react to well identified triggers and, hence, adapt themselves to changes brought about on their state or configuration.

For instance, in the banking domain that we have been using as an example, a coordination context normally exists for each customer. The purpose of this context is to manage the relationships that the customer may hold with its various accounts according to the packages that the bank offers. 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 ATMs.

The syntax that we are developing for contexts can be illustrated as follows:

```
coordination context customer(c:customer)
workspace
  component types  account, customer
  contract types standard-withdrawal, VIP-withdrawal,
                    pensioner-package, home-owner-package
constants min-VIP: money
attributes  avg-balance = …
services
  subscribe_VIP(a:account,V:money):
        pre:   c.owns(a) & avg-balance•min-VIP &
                    not exists home-owner-package(c,a)
        post:  exists' VIP-withdrawal(c,a) &
                    VIP-withdrawal(c,a)'.credit=V
  subscribe_home(a:account):
        pre:   not exists pensioner-package(c)
        post:  c.owns(a)' & exists' home-owner-package(c,a)
  subscribe_pensioner:
        pre:   not exists pensioner-package(c) &
                    not exists home-owner-package(c,a)
        post:  exists' pensioner-package(c)
rules
  VIP-to-std:
        when exists VIP-withdrawal(c,a) & avg-balance<min-VIP
        post not exists' VIP-withdrawal(c,a) &
                    exists' standard-withdrawal(c,a)
end context
```

Besides the laws that we introduced in the previous sections for managing withdrawals, we have added the names of a few other ones just to make the example more “interesting”. These account for other financial packages such as those concerned with retirement pensions and credit for home-purchase. We leave their specification to the reader as an exercise!

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). Under “workspace” we identify the component and contract types (laws) that are made available for evolving the way the anchor interacts with the rest of the system.

Configuration services correspond to operations for ad-hoc reconfiguration, i.e. they are performed on demand from users of the system. Notice that we include in this category operations that, in traditional OO modelling, are assigned to classes like object creation. The rationale is that, by interfering with the population of the system, such operations address the evolution of its configuration and, hence, their use should be regulated in the scope of a coordination context. Configuration services involve both components and contracts. In the example above, besides the creation of new accounts, three other services are provided for each of the contracts that models a financial package that can be offered to the customer. These services have preconditions through which business policies are enforced. For instance, VIP-withdrawals are not available on accounts that support a home-owner package; pensioners are not allowed to subscribe home-owner packages.

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. In the example above, a VIP-package is replaced by a standard one when the average balance of the customer falls below the minimum value set up for being a VIP. 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.

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 interfaces from their implementations, which adds to flexibility by allowing the choice of the actual reconfiguration operations to depend on “lower level” issues like the physical distribution topology. The pre/post-conditions capture business policies that should be elicited during analysis, like eligibility conditions as illustrated in the example, or dependencies that regulate the subscription of different business products, as well as legislation that becomes applicable, etc. When writing post-conditions, we use primed expressions to denote the value that they take in the state that is obtained by executing the service or rule.

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, 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.

Each context provides a set of business services that, taken together, form a “profile” of business activity. Such services are implemented using the configuration layer (to query and change the current configuration) and the functional operations provided by components and contracts. Since the environment can only access and manipulate the system through the provided contexts, i.e., the bottom three layers are not visible to outside the system, contexts set constraints on the nature of the operations that can be performed. These constraints add to whatever policies, rules, and invariants have been declared for the configuration, contracts, and components, and should reflect properties that are specific to a business context. For example, the withdrawal service in the ATM context has a fixed limit on the amount that can be withdrawn, irrespective of the balance in the customer account. This constraint will be added to whatever business rules are in effect for withdrawals performed by that customer, because the service will call directly the debit operation on the account, which in turn will trigger whatever contracts are in effect.

Indeed, although the example above does not illustrate it, contexts can also make available operations that act on the states of the components through the methods that these offer through their public interfaces. Different contexts may even make use of different implementations for the same operations, for instance reflecting the fact that access to the system may be provided through different channels. For instance, the withdrawal service is typically offered in a coordination context that models access over the counter at the local branch, but not if the access is via the Internet, whereas the amount that can be requested will be limited for accesses via an ATM.

Contexts can be used to model actors as in use cases [9], 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. 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 context as a modelling 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,

From a methodological point of view, contexts become necessary, and come into play, during the transition from a business model to a conceptual/logical model of the intended software system. This should be the first time in the process that a boundary starts to be drawn between the information system and the business environment, leading to the need for deciding on the interface in which it will be managed.

Concerning the relationship between coordination contexts and business policies, for simplicity, we take policies to be cast as properties of system configuration that are required to be invariants of the evolution process. The language that we use for representing business invariants is a first order language over the observable attributes of components (methods that return values) extended with a predicate **exists** that indicates whether a given component or contract instance is part of the current configuration, i.e. of the subsystem defined by the context. Examples of business policies expressed in this language are "**forall** a:account, **not** (**exists** VIP(c,a) **and exists** home-owner(c,a))" and "**forall** a:account, **not** (**exists** home-owner(c,a) **and exists** pensioner(c))". Both these properties can be proved to be invariants for the coordination context that we defined above. Once a property is proved to be an invariant of every coordination context defined for the application, it will be an invariant for the whole system, meaning that it will emerge as a property of every configuration that can be reached during the evolution of the system.

# 5 Tool Support

There are two important aspects to discuss as far as automation and tool support for the proposed architectural approach is concerned. On the one hand, the nature of an environment in which development can be supported. On the other hand, specific techniques that support the coordination-based approach over platforms that do not offer first-class primitives for implementing coordination rules.

## 5.1 Architecture of an Environment for Coordination

Using an architectural-based approach for software development requires new activities to be integrated in the process. An environment that supports the CCC-approach should have a tool dedicated to the coordination layer development and, at the same time, providing support for run-time reconfiguration. Such a tool may be autonomous or integrated with the rest of the environment by one of the several integration techniques currently available. In this section, we describe the main characteristics of an environment that we have been developing – the CDE (Coordination Development Environment) – as a proof of concept and prototype for testing the viability of a larger and more comprehensive coordination-based support to software development.

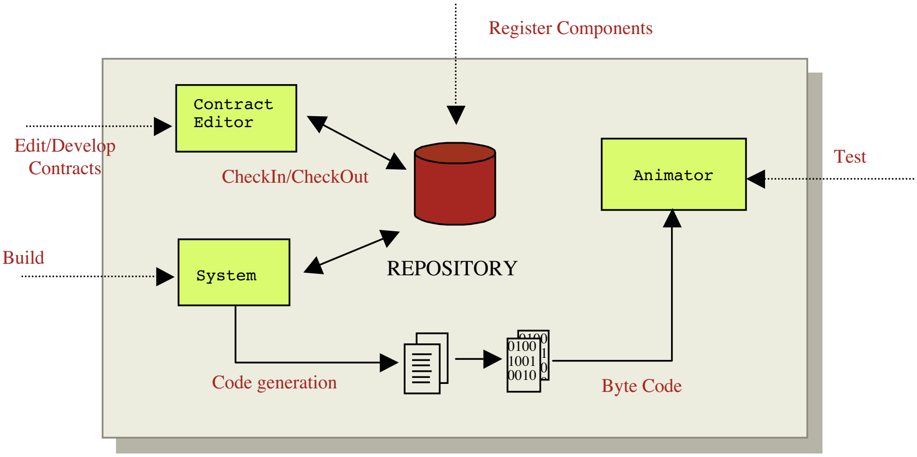
The main functionality of the CDE is to allow the definition of contract types (laws) and provide the necessary deployment of the final pieces of the system. It is assumed that the components to be coordinated are somehow available, developed in another part of the environment, or given by the developer. The CDE should provide facilities not only for the development/testing of coordination, but also run-time facilities for the management of contracts such as manual creation/deletion of contracts between objects, configuration of policies for automatic creation of contracts and priorities among others. The logical architectural components of the CDE, namely the Editor, the Repository, the Builder and the Animator, that support the previous activities, are shown below.

At the development level, the CDE provides the following functionalities:

- *Registration*: components are registered to the tool as candidates for coordination.
- *Edition*: Contract types (laws) are defined connecting some registered components. Coordination rules and constraints are defined on those contracts.
- *Deployment*: the code necessary to implement the coordinated components and the contract semantics in the final system is produced by *generating* the necessary parts according to a contract micro-architecture we have developed (see section 5.2). This micro-architecture is based on the Proxy and Chain of Responsibility design patterns and handles the superposition of the coordination contracts over existing components in a way that is transparent to the component and contract designer. The micro-architecture allows coordination contracts to be directly implemented using OO languages, fulfilling the following very important required properties:

- components are not aware of the presence of contracts and, therefore, any number of contracts can be added/removed without having to modify the components,
- contracts can be added/deleted in a "plug and play" mode, even at run time,
- even existing components can be easily adapted to accept contracts without making any modifications elsewhere in the application, thus allowing for easier reengineering/evolving of existing applications.

- *Animation*: facilities are provided allowing testing/prototyping of contract semantics. This layer provides a simple way of creating a system configuration and triggering operations on application objects, observe their state, and the dynamics associated with the existence of contracts at work.



The architectural concepts described herein may be applied to different levels during system development depending on several factors, such as the characteristics of the components, the way components are built, the development phase where the coordination concept is going to be used, among others. One of the main motivations in the development of the modelling primitives discussed in the previous sections was to provide coordination as a top-level abstraction in development, right from the modelling phases. By using these concepts, one can provide a clear separation between computational concerns and the communication aspects, leading to flexible and clear models. Current modelling techniques that are used in industry, even state-of-the-art such as UML, do not provide any similar concept. Therefore, to be more effective in a working environment, a tool providing coordination facilities may be integrated with existing UML modelling tools. However, developers may choose to work with the contract coordination concept at a more detailed design/implementation level, where coordination aspects do not appear as modelling artefacts but, rather, as implementation constructs that provide more flexibility in terms of evolution. These two contexts of use can be summarised as follows:

- *Model Coordination:* Coordination is used at the Analysis or Design phases. Components are model classes (e.g. UML classes). Coordination contracts make a Coordination Model on top of the Analysis/Design Model. The deployment activity must take into account the way final coded components are obtained from model components and provide the necessary integration.
- *Construction Coordination:* Coordination is used in the construction phase. Components are the final coded components of the basic building blocks of the system. Coordination contracts are defined directly over implementation classes. It is suitable to be applied on the evolution of an existing system. The first version of CDE we have already developed works in such a context with Java as the target deployment language and animation capabilities based on sequence diagrams.

It should be clear, however, that the type of components under potential coordination may define the context and capabilities in which the CDE is used. Furthermore, the specific language and technical environment may impose constraints on the coordination features that can be used because techniques to achieve the implementation of its semantics may not be available. For instance, if Java is the target deployment language, synchronisation sets (the actions of all the coordinations active) may not be fully transactional in the sense that there is no rollback associated, when a failure or exception occurs. This is due to the semantics of the underlying language (Java). In other words, it should be clear that the output is strongly dependent on several other environmental conditions.

Therefore, in order to effectively use coordination technology in the development process, two main aspects must be clearly defined first: where to use it, and what are the building blocks that can be put under coordination. On the other hand, the real effectiveness of coordination laws as a semantic primitive for conceptual modelling depends on the availability of support tools such as CDE, which provide automatic code generation from higher-level specifications. Such tools hide the implementation complexity of coordination, allowing the developer just to handle the law itself. But, even in the absence of such tools, we feel that coordination laws provide a useful abstraction mechanism to be used in conceptual modelling because they direct developers to the identification and promotion of interactions as first-class citizens, a precondition for taming the complexity of system construction and evolution.

### 5.2 A Micro-Architecture for Coordination

As already explained in previous sections, a contract (instance of a law) works as an active agent that coordinates a number of partners (components that instantiate the coordination interfaces of the law). In this section, we are concerned with the way these coordination mechanisms can be implemented in general and the specific way in which they are handled in the CDE.

When defining an implementation, we need to have in mind that, as motivated in the introduction, we should be able to superpose a contract to given objects in a system and coordinate their behaviour as intended *without having to modify the way these objects are implemented.* This degree of flexibility is absolutely necessary when the implementation of these objects is not available or cannot be modified, as in legacy systems. It is also a distinguishing factor of contracts when compared with existing mechanisms for modelling object interaction, and one that makes contracts particularly suited in business domains where the ability to support the definition and dynamic application of new forms of coordination is a significant market advantage.

Different standards for component-based software development have emerged in the last few years. However, none of these standards provide a convenient and abstract way of supporting coordination as a first-class mechanism. Because of this, we propose our solution as a micro-architecture that exploits some widely available properties of object-oriented programming languages such as polymorphism and subtyping, and is based on other well known design patterns, namely the Broker, and the Proxy or Surrogate [15].

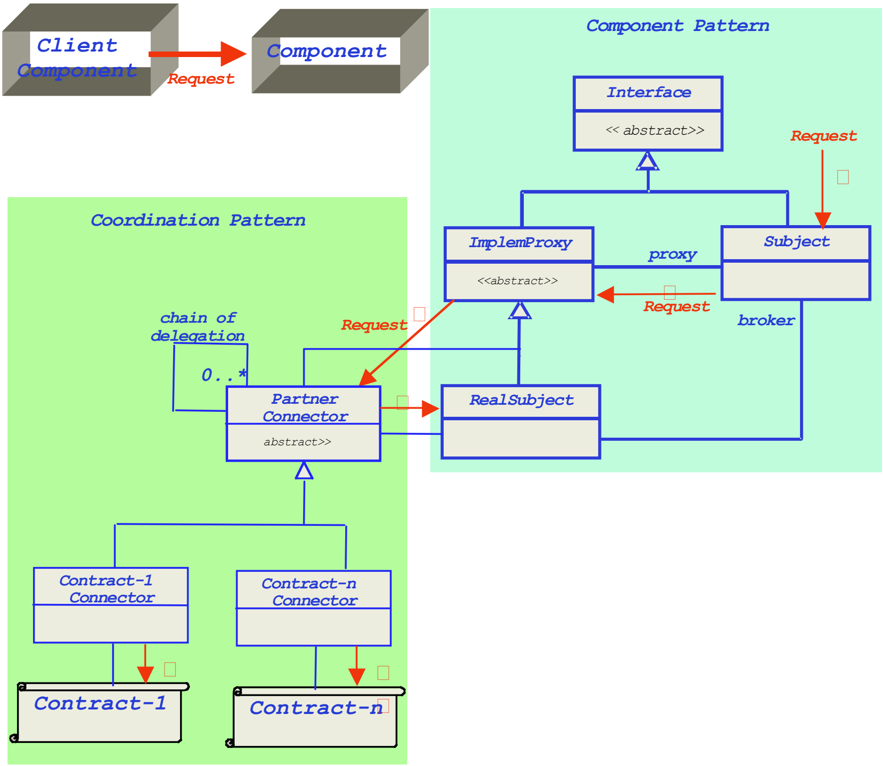
It is important to stress that what we are proposing is just a possible solution for implementing contracts, a concrete evidence that the concept can be made effective in our working environments. Other patterns may well be used and other mechanisms may very well prove to be more useful for capturing the required coordination effects in specific development platforms. For instance, the use of reflection is an alternative that looks promising enough to be explored.

More specifically, our aim in the paper is not to promote the use of the specific patterns that we are going to show: these patterns have existed for a long time, and other patterns exist that fulfil the same purpose. Our contribution is in the modelling primitives that we are proposing for supporting the externalisation of interactions as first-class citizens in conceptual models. The patterns are not the instrument that will assist conceptual modelling: coordination laws are.

The proposed micro-architecture is depicted below in terms of two main structures. One of them, what we have called the *component pattern*, consists of the features that have to be provided for each component so that it can become coordinated by a contract. In a nutshell, these features support the externalisation of the interface of each component as formalised in [13]. They provide a specific interface *(Interface)*, as an abstract class, for every component. This interface is linked to the real program *(RealSubject)* through a dynamically reconfigurable proxy reference. The classes that participate in the component pattern can be summarised as follows:

*Interface* – an abstract class that defines the common interface of services provided by *ImplemProxy* and *Subject.*

*Subject* – a broker (concrete class) maintaining a reference that lets the subject delegate received requests to *ImplemProxy* using the polymorphic entity proxy. At run-time, it may point to a *RealSubject* if no contract is involved, or point to a *PartnerConnector* that links the real subject to the contracts that coordinate its behaviour.



*ImplemProxy* – an abstract class that defines the common interface of *RealSubject* and *PartnerConnector*. The interface is inherited form *Interface* to guarantee that all these classes offer the same interface as *Subject* (the broker) with which real subject clients have to interact. It is also extended with a new operation that can be redefined in *RealSubject* and *PartnerConnector* to monitor state changes in the partner represented by *RealSubject*. The implementation of this operation in *PartnerConnector* delegates to each contract the execution of the synchronisation sets that are triggered by conditions on the monitored state.

*RealSubject* – the concrete domain class with the business logic that defines the real object that the broker represents. The concrete implementation of provided services is in this class.

The second structure in the micro-architecture concerns the mechanisms that coordinate the given component as achieved through the contracts that are in place for that component. The classes involved in this pattern are as follows.

*PartnerConnector* – maintains the connection between the real object (*RealSubject*) and the contracts in place for it. Adding or removing contracts to coordinate that real object does not require the creation of a new instance of this class but only of a new association with the new contract and an instantiation link with the existing instance of *PartnerConnector*. This means that there is only one instance of this class associated with one instance of *RealSubject*. A chain of delegation, also depicted in the diagram, controls the way the different contracts superpose their own coordination mechanisms. The *PartnerConnector* offers an interface that includes all the operations offered by *Subject/RealSubject*. The implementation of these operations is delegated on *RealSubject*. According to this strategy, calls to this pattern that are included in the synchronisation set are delegated to either *Subject* or *RealSubject*. The former occurs when the call does not refer to the operations under coordination, i.e. when the called operation is not the trigger of the current coordination.

*Contract* – a coordination object that is notified and takes decisions whenever a request is invoked on a real subject.

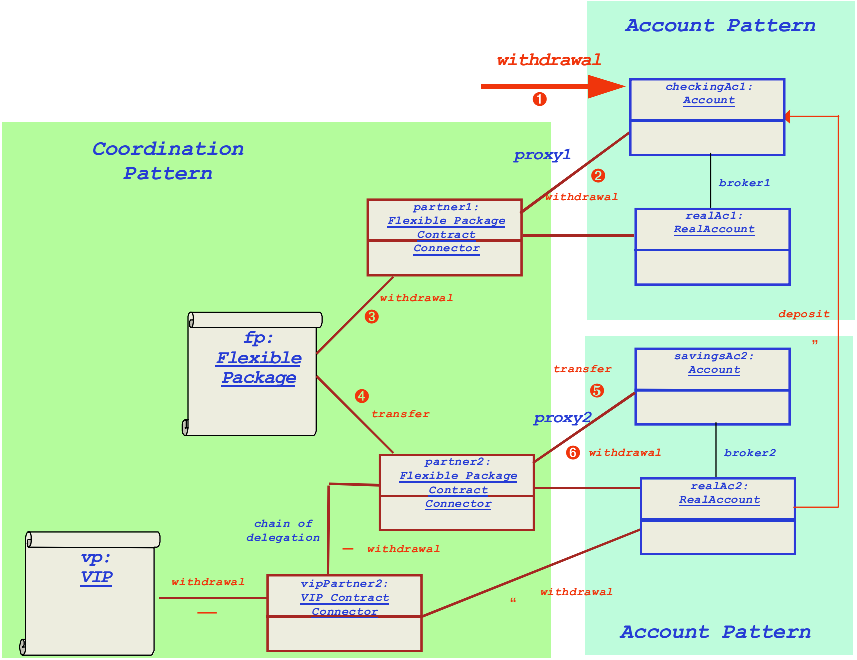
The *chain of delegation* is the mechanism that regulates the way in which the different contracts that apply to the component interact with each other. Typically, this chain of delegation will establish a priority among these contracts. An example will be given below.

Support for dynamic reconfiguration of the code executed upon requests for operations, including requests by *self* as in active objects, is achieved through the proxy. Reconfiguration of a predefined component (such as adapting the component for a specific use) or coordination of various components (such as behaviour synchronisation) is achieved by making the proxy reference a *polymorphic entity*. On the one hand, this proxy is provided with a *static type* at compile-time – the type with which this entity is declared (*ImplemProxy*) – that complies with the interface of the component. On the other hand, the type of its values may vary at run-time through *PartnerConnector* as connectors are superposed or removed from the configuration of the system. These types, the ones that can be dynamically superposed, become the entity's *dynamic type* (dynamic binding).

The notion of dynamic binding means that, at run-time, such a proxy assumes different values of different types. However, when a request is made for services of the component, it is the dynamic binding mechanism of the underlying object-oriented language (e.g. C++, Java) that makes the choice of the operation implementation to execute (*RealSubject* or *PartnerConnector*) based on the type to which the proxy belongs.

In order to explain, in more detail, how coordination is handled by the proposed pattern, consider a specific example: two accounts – a checking account and a savings account – under the coordination of a Flexible Package, such that the savings account is also subject to a VIP-package. The figure below depicts the chain of actions among the concrete classes involved in the pattern that are triggered by a withdrawal request received by the checking account.

The request is handled by the broker of the target of the request (the checking account) and, through the proxy, is delegated on the FlexiblePackage connector. This connector handles the request as specified in the previous section: it transfers funds from the other partner – the savings account – if the current balance of the checking account is not enough to satisfy the request. If such a transfer is necessary, the withdrawal from the savings account falls under the coordination of the VIP-connector. The diagram depicts a situation in which the chain of delegation controlling the contracts on the savings account gives priority to the flexible package connector, which then delegates on the VIP connector because the withdrawal action on the savings account is not under the coordination of the flexible package. The final withdrawal, the effective one, is performed on the real savings account after having been cleared by the VIP-connector. All this is done atomically. In particular, if the VIP-connector refuses the request for a withdrawal from the savings account, the original withdrawal sent to the checking account is also refused and any alterations made in the meanwhile on the state of the objects involved are undone.



Notice that, as explained above, there is only one instance of *PartnerConnector* connected to the proxy. It is important to notice that before the partner connector gives rights to the real object implementation to execute a request, it intercepts the request and gives right to the contract to decide if the request is valid and perform other actions. This interception allows us to impose other contractual obligations on the interaction between the caller and the callee. On the other hand, it allows the contract to perform other actions before or after the real object executes the request. On the other hand, if there are no contracts coordinating a real subject, the only overhead imposed by the pattern is an extra call from the broker to the real object. We believe that this is a small price to pay for the added flexibility that the pattern brings to system evolution as a whole.

## 6 Concluding Remarks

In this paper, we presented a set of techniques that, in our opinion and from our experience, can supplement the shortcomings of object-oriented approaches in endowing systems with the levels of agility required for operating in "internet-time" and support the next generation of the e-world – Web Services, B2B, P2P…

Basically, we argued that the move from an "identity" to a "service"-oriented approach, replacing the tight coupling that explicit feature calling – the basic mechanism of OO-computation – imposes on systems in favour of interconnections that can be established "just-in-time", can be supported by clearly separating computation and coordination concerns and relying on the superposition of external connectors for establishing interactions between components that are otherwise completely unaware of one another. Reconfiguration techniques as known from Distributed Systems can then be used for addressing system evolution, namely the process of dynamically interconnecting components according to the business rules that determine how the system should behave at each time.

Although the paper focused on the semantic primitives – coordination contracts, laws and contexts – their intuitive semantics, and the tool support that can be provided for their application, our approach is well supported at the foundational level through a mathematical characterisation of the notions of coordination, superposition and reconfiguration [e.g.13,43].

### 6.1 Related Work

We acknowledged several times already the contributions that we borrowed from the areas of Coordination Languages and Models, Software Architectures, and Reconfigurable Distributed Systems. We have also mentioned that several other authors have made similar observations about the need to make explicit and available, as first-class citizens, the rules that govern the behaviour of systems, namely in [22], which became the subject matter of the ISO General Relationship Model (ISO/IEC 10165-7). The semantic primitives that we proposed in the paper are in the spirit of this work but add to it the evolutionary aspects that they inherit from the architectural approaches, and the concurrency and synchronisation aspects that they inherit from Distributed System Design. A recent evolution of this approach that we intend to investigate in relationship to ours is the treatment of roles (and templates) in RM-ODP [21].

Another notion of contract can also be found in [7] that emerged in the context of the action-systems approach to parallel program design [6]. Like in our approach, it promotes the separation between the specification of what actors can do in a system and how they need to be coordinated so that the computations required of the system are indeed part of the global behaviour. However, the architectural and evolutionary dimensions are not explored as such.

Concrete extensions to the UML in order to provide automated support for evolution are proposed in [31] through "evolution contracts". The idea behind evolution contracts is that incremental modification and evolution of software artifacts is made explicit by means of a formal contract between the provider and the modifier of the artifact. The purpose of the evolution contract is to make evolution more disciplined. The coordination contracts that we presented can also extend the UML but at the level of the semantic primitives that can be used for modelling, facilitating evolution, not managing it. For managing evolution, we provide explicit primitives at the configuration layer as already discussed.

Besides these related notions of contract, it is important to mention the work of N.Minsky and his colleagues who, for quite some time, have been developing what they call Law-Governed Systems/Architectures/Interactions (e.g. [33]). However, the emphasis is put more on the architectural aspects of system design (e.g. as enforcing a token-ring architecture throughout evolution) whereas we tend to focus more on policies and other properties that relate to the business level. This is a rather simplistic and reductionist comparison because, on the one hand, we have also explored our coordination technologies for design and technical architectures and, on the other hand, Minsky's work seems to be applicable to the earlier levels of development as well. Hence, the truth is that more research is necessary to investigate how the two approaches actually relate and can benefit from the experience that both groups have developed in applying them.

### 6.2 Future Perspectives

The methodology and language that supports the definition of policies (e.g. as invariants) and contexts is still very much in its infancy but quickly progressing, mainly through the application of the approach to a number of case studies [e.g.4,23,24]. Even at this early stage, it is clear that coordination contexts provide an effective means of defining the way in which the evolution process can be controlled and the enforcement of business policies automated. We are now extending tool-support [5,17] and logical analysis [12] to this particular activity. Part of this effort is being planned as a collaboration initiative with Leonor Barroca at the Open University and Kevin Lano at King's College London.

Work is progressing in several other fronts, including the extension of the architectural approach to distribution and mobility in the IST project 2001-32747 (AGILE – Architectures for Mobility) in which ATX Software and the University of Leicester are partners together with IEI/CNR and the Universities of Florence, Lisbon, Munich, Pisa and Warsaw. Preliminary results can be found in [4,28]. Applications to data-intensive systems are also being planned as part of a research collaboration with the CERN. The integration of the method, semantic modelling primitives, and support tools with OMG activities around the UML is being planned in collaboration with a team of researchers coordinated by A.Moreira at the New University of Lisbon.

## Acknowledgements

The material included in this paper is the result of a research effort that, over several years, has involved many people, both in industry and academia. We are deeply grateful to the following people in particular, for their continued contribution and dedication to this project: J.Gouveia, G.Koutsoukos, A.Lopes, and M.Wermelinger,

## References

1. R.Allen and D.Garlan, "A Formal Basis for Architectural Connectors", *ACM TOSEM,* 6(3), 1997, 213-249.
2. L.F.Andrade and J.L.Fiadeiro, "Interconnecting Objects via Contracts", in R.France and B.Rumpe (eds), *UML'99 – Beyond the Standard*, LNCS 1723, Springer Verlag 1999, 566-583.
3. L.F.Andrade and J.L.Fiadeiro, "Service-Oriented Business and System Specification: Beyond Object-orientation", in H.Kilov and K.Baclwaski (eds), *Practical Foundations of Business and System Specifications*, Kluwer Academic Publishers 2003, 1-23.
4. L.F.Andrade, J.L.Fiadeiro, A.Lopes and M.Wermelinger, "Architectural Techniques for Evolving Control Systems", in *Formal Methods for Railway Operation and Control Systems*, G.Tarnai & E.Schnieder (eds), L'Harmattan Press, 2003
5. L.F.Andrade, J.Gouveia, G.Koutsoukos and J.L.Fiadeiro, "Coordination Contracts, Evolution and Tools", *Journal on Software Maintenance and Evolution: Research and Practice* 14(5), 2002, 353-369.
6. R.Back and R.Kurki-Suonio, "Distributed Cooperation with Action Systems", *ACM TOPLAS* 10(4), 1988, 513-554.
7. RJ.Back, L.Petre and I.Paltor, "Analysing UML Use Cases as Contracts", in *UML'99 – Beyond the Standard,* R.France and B.Rumpe (eds), LNCS 1723, Springer Verlag 1999, 518-533.
8. L.Bass, P.Clements and R.Kasman, *Software Architecture in Practice*, Addison Wesley 1998.
9. G.Booch, J.Rumbaugh and I.Jacobson, *The Unified Modeling Language User Guide,* Addison-Wesley 1998.
10. T.Elrad, R.Filman and A.Bader (Guest editors). Special Issue on Aspect Oriented Programming. *Communications of the ACM.* October 2001; 44(10)
11. M.Endler and J.Wei, "Programming Generic Dynamic Reconfigurations for Distributed Applications", in *Proc. 1st Intl. Workshop on Configurable Distributed Systems*, 1992, 68-79.
12. J.L.Fiadeiro, N.Martí-Oliet, T.Maibaum, J.Meseguer and I. Pita, "Towards a Verification Logic for Rewriting Logic", in *Recent Trends in Algebraic Development Techniques*, D.Bert and C.Choppy (eds), LNCS 1827, pp. 438-458, Springer-Verlag 2000
13. J.L.Fiadeiro, A.Lopes and M.Wermelinger, "A Mathematical Semantics for Architectural Connectors", in *Generic Programming*, R.Backhouse and J.Gibbons (eds), LNCS, Springer-Verlag, in print.
14. P.Finger, "Componend-Based Frameworks for E-Commerce", Communications of the ACM 43(10), 2000, 61-66.
15. E.Gamma, R.Helm, R.Johnson and J.Vlissides, *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley 1995
16. D.Gelernter and N.Carriero, "Coordination Languages and their Significance", *Communications ACM* 35(2), 1992, 97-107.
17. J.Gouveia, G.Koutsoukos, L.Andrade and J.Fiadeiro, "Tool Support for Coordination-Based Software Evolution", in *Technology of Object-Oriented Languages and Systems – TOOLS 38,* W.Pree (ed), IEEE Press 2001, 184-196.
18. P.Herzum and O.Sims, *Business Component Factory*, Wiley 2000.
19. J.Hopkins, "Component Primer", *Communications of the ACM* 43(10), 2000, 27-30.

20. W.Kent, “Participants and Performers: A Basis for Classifying Object Models”, in *Proc. OOPSLA 1993 Workshop on Specification of Behavioral Semantics in Object-Oriented Information Modeling*, 1993
21. H.Kilov, *Business Models*, Prentice-Hall 2002.
22. H.Kilov and J.Ross, *Information Modeling: an Object-oriented Approach*, Prentice-Hall 1994.
23. G.Koutsoukos, J.Gouveia, L.Andrade and J.L.Fiadeiro, “Managing Evolution in Telecommunications Systems”, in *New Developments on Distributed Applications and Interoperable Systems*, K.Zielinski, K.Geihs and A. Laurentowski (eds), Kluwer Academic Publishers 2001; 133-139.
24. 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”, in R.Corchuelo, A,Ruiz and M.Toro (eds), *Advances in Business Solutions*, Catedral Publicaciones 2002, 45-56.
25. J.Kramer, "Configuration Programming – A Framework for the Development of Distributable Systems", in *Proceedings CompEuro'90*, IEEE Computer Society Press 1990; 374-384.
26. J.Kramer, "Exoskeletal Software", in *Proc. 16th ICSE*, 1994, 366.
27. K.Lano, J.Fiadeiro and L.Andrade, *Software Design in Java 2*, Palgrave-Macmillan, 2002.
28. A.Lopes, J.Fiadeiro and M.Wermelinger, "Architectural Primitives for Distribution and Mobility", *SIGSOFT 2002/FSE-10*, ACM Press 2002, 41-50.
29. J.Magee and J.Kramer, "Dynamic Structure in Software Architectures", in *4th Symp. on Foundations of Software Engineering*, ACM Press 1996, 3-14.
30. N.Medvidovic and R.Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages", *IEEE Trans. on Software Eng.,* 26(1), Jan. 2000, 70-93.
31. T.Mens and T.D’Hondt, "Automating Support for Software Evolution in UML", in *Automated Software Engineering Journal* 7, Kluwer Academic Publishers, 2000, 39-59.
32. B.Meyer, "Applying Design by Contract", in *IEEE Computer* (25)10, 1992, 40-51.
33. N.Minsky and V.Ungureanu, “Law-Governed Interaction: A Coordination & Control Mechanism for Heterogeneous Distributed Systems” in *ACM TOSEM* 9(3), 2000, 273-305.
34. K.Moazami-Goudarzi, *Consistency Preserving Dynamic Reconfiguration of Distributed Systems*, PhD Thesis, Imperial College London, 1999.
35. R.Monroe, *Capturing Software Architecture Design Expertise with Armani*, Tech. Rep. CMU-CS-98-163, School of Computer Science, Carnegie Mellon University, Oct. 1998.
36. D.Notkin, D.Garlan, W.Griswold and K.Sullivan, “Adding Implicit Invocation to Languages: Three Approaches”, in *Object Technologies for Advanced Software*, S.Nishio and A.Yonezawa (editors), LNCS 742, Springer-Verlag 1993, 489-510.
37. P.Oreizy, N.Medvidovic and R.Taylor, "Architecture-based Runtime Software Evolution", in *Proc. ICSE’98*, IEEE Computer Science Press 1998
38. D.Perry and A.Wolf, "Foundations for the Study of Software Architectures", *ACM SIGSOFT Software Engineering Notes* 17(4), 1992, 40-52.
39. M.Snoeck, G.Dedene, M.Verhels and A-M.Depuydt, *Object-oriented Enterprise Modelling with MERODE*, Leuvense Universitaire Press, 1999.
40. M.Shaw, "Procedure Calls are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status", in D.A. Lamb (Ed.), *Studies of Software Design*, LNCS 1078, Springer-Verlag 1996.
41. M.Shaw and D.Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall 1996.

42. C.Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison Wesley 1998.

43. M.Wermelinger, A.Lopes and J.L.Fiadeiro, "A Graph Based Architectural (Re)configuration Language", in *ESEC/FSE'01*, V.Gruhn (ed), ACM Press 2001, 21-32.