Coordination Contracts, Evolution and Tools

L.Andrade, J.Gouveia, G.Koutsoukos
Oblog Software S.A.
Alameda Antonio Sergio 7-1A
2795 Linda-a-Velha, Portugal
{landrade, jgouveia, gkoutsoukos}@oblog.pt

Jose Luiz Fiadeiro
Department of Informatics
Faculty of Sciences, University of Lisbon
Campo Grande, 1700 Lisboa, Portugal
jose@fiadeiro.org

Abstract

In this paper, we propose the adoption of coordination contracts – a modeling primitive grounded on a formal semantics based on the Category Theory – as a basis for a discipline that effectively supports software evolution. We give an overview of coordination contracts, present examples of how they can support evolution, and describe a development environment through which they can be used in practice.

1. Introduction

No one can admittedly deny the fact that today we are witnessing tremendous technological advances in a highly competitive business environment. To the question whether technology is forming business or vice-versa, organisations are replying by integrating their business and IT strategies, thus using technology to do business. As a result, there is an increasing pressure for building software systems that are able to cope with new requirements imposed by both technological advances and different business rules. Even worse, as a result of e-economics, systems often have to be able to accommodate changes in run-time, even performed directly by customers. As a consequence, organisations are seeking answers on how to conceive and develop systems that are adaptive to change.

For better or for worse, organisations are looking for solutions to this problem in the context of object-oriented development techniques such as the UML, and component-based frameworks such as COM and CORBA. However, as explained in [1,2] experience has shown that the benefits that object-oriented techniques have brought to software *construction* cannot be extended directly to software *evolution*. Moreover, disciplines that, in theory, support software evolution, in practice often fail to provide a means for their implementation and, unfortunately, end up being buried in the literature. As a consequence, it is not surprising that existing tools that intend to offer support for evolution are far from ideal.

In this paper, we propose the adoption of the coordination contract modelling primitive presented in [1,2,4], grounded on a formal semantics based on Category Theory [2,4], as a basis for a discipline that can lead to software systems that are adaptive to change. We briefly discuss coordination contracts and we present examples from banking and telecommunications on how contracts can support software evolution. Moreover, we present and discuss the scope, applicability and impact on the development life-cycle of an environment that we have been building for allowing coordination contracts to be effectively used as a technology.

2. Coordination Contracts

As discussed in [1,2] the rationale for the definition of coordination contracts is the realization that, in highly volatile domains, one can distinguish between two different kinds of entities as far as the evolution of the application domain is concerned. On the one hand, there are classes of objects that correspond to business entities that are relatively stable in the sense that they capture core concepts or “invariants” of the domain. On the other hand, we need objects that have to keep changing in order for the system to reflect the dynamics of the application

domain. These require a layer of coordination to be established over the functionalities of the stable entities so that the behavior that is required from the system can emerge, at each state, from the interconnections that this layer of coordination puts in place.

These coordination aspects need to be made available explicitly in the system models so that they can be changed, as a result of modifications occurring at the level of requirements, without affecting the basic objects that compose the system. The purpose of contracts is to provide mechanisms for that layer of coordination to be modeled and implemented in a compositional way.

In general terms, a coordination contract is a connection that is established between a group of objects where rules and constraints are superposed on the behavior of the participants, thus enforcing a specific form of interaction. From a static point of view, a contract defines what in the UML is known as an *association class*. However, the way interaction is established between the partners is more powerful than what can be achieved within the UML and similar OO languages because it relies on the mechanism of *superposition* as developed for parallel and distributed system design [5,6,8]. When a call is made from a client object to a supplier object, the contract "intercepts" the call and *superposes* whatever forms of behavior it prescribes. In order to provide the required levels of pluggability, neither the client, nor any other object in the system, needs to know what kind of coordination is being superposed. To enable that, a contract design pattern, presented in [3,7], allows coordination contracts to be superposed on given objects in a system to coordinate their behavior without having to modify the way the objects are implemented.

Coordination Contracts are currently supported by a specification language called Oblog, but the underlying technology is independent of the language. In Oblog notation, a coordination contract is defined as follows:

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

The classes of objects that are related by the contract are identified under *participants*. A contract may also specify constraints that represent invariants defining in which conditions instances from the participating classes may be related by the contract, attributes and operations private to the contract, and the prescription of the coordination effects that will be superposed on the partners. Each interaction under a coordination rule is of the form:

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

The name of the interaction is used for establishing an overall coordination among the various interactions and the contract's own actions. The condition under "when" establishes the trigger of the interaction. The trigger can be a condition on the state of the participants, a request for a particular service, or a signal received by one of the participants. Several conditions can be placed in the "when" clause using the keyword "AND". If one of such conditions is not satisfied, the contract is considered as being "inactive" and, as a result, either the original code of the trigger or another contract is executed. This mechanism provides the ability for controlling which of the contracts imposed on a component will be responsible for coordinating it.

The "do" clause identifies the reactions to be performed, usually in terms of actions of the partners and some of the contracts own actions. When the trigger corresponds to an operation, three types of actions may be superposed on the execution of the operation:

1. **before**: to be performed before the operation
2. **replace**: to be performed instead of the operation
3. **after**: to be performed after the operation

In the case in which an object participates in multiple contracts with the same trigger, the sequence of execution for the before, replace and after clauses is shown in Figure 1. It should be noted that the semantics of contracts allow for only one "replace" clause to be executed, thus preventing the undesirable situation of having two alternative actions for the same trigger. Furthermore, any such replacement action must adhere to whatever specification clauses apply to the operation (e.g. contracts in the sense of Meyer [9] specifying pre- and post-conditions). This ensures that the functionality of the original operation, as advertised through its specification, is preserved.

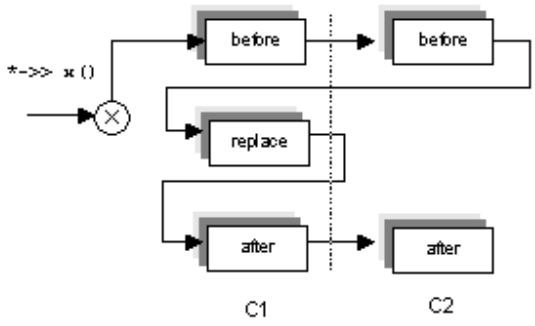

Figure 1. Execution of multiple contracts

The actions that are executed as part of the "do" clause are called the synchronization set associated with the trigger. The semantics of contracts require that this set be executed atomically, guarded by the conjunction of the guards of the individual actions together with the conditions included in the "with" clause. Therefore, the "with" clause puts further constraints on the execution of the actions involved in the interaction. If any condition under the "with" clause is not satisfied, an exception is thrown as a result and none of the actions in the synchronization set is executed.

For a detailed description of coordination contracts and their formal semantics, the reader is urged to consult [2,4]. In what follows we present an example from banking to motivate the scope and solutions coordination contracts can offer. Consider a world of bank accounts in which clients can, as usual, make withdrawals. The object class *account* is usually specified with an attribute *balance* and a method *withdrawal* with parameter *amount*. In a typical implementation one can assign the guard *Balance>=amount* restricting this method to occur in states in which the amount to be withdrawn can be covered by the balance. However, as explained in [1] assigning this guard to *withdrawal* can be seen as part of the specification of a business requirement and not necessarily of the functionality of a basic business entity like account. Indeed, the circumstances under which a withdrawal will be accepted can change from customer to customer and, even for the same customer, from one account to another depending on its type. As discussed in [1] inheritance is not a good way of changing the guard in order to model these different situations. Firstly, inheritance views objects as white boxes in the sense that adaptations like changes to guards are performed on the internal structure of the objects, which from the evolution point of view of is not desirable. Secondly, from the business point of view, the adaptations that make sense may be required on classes other than the ones in which the restrictions were implemented. In our example, this is the case when it is the type of client, and not the type of account, that determines the nature of the guard that applies to withdrawals. The reason the guard will end up applied to *withdrawal*, and the specialisation to *account,* is that, in the traditional clientship mode of interaction, the code is placed on the supplier class. Therefore, it makes more sense for business requirements of this sort to be modeled explicitly outside the classes that model the basic business entities, because they represent aspects of the domain that are subject to frequent changes (evolution). Our proposal is that guards like the one discussed above should be modeled as *coordination contracts* that can be established between clients and accounts. For instance, consider the following contract that allows for using the functionality of *withdrawal* to relax the situations in which an account may be overdrawn.

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

To further illustrate how contracts can be applied to support the evolution of requirements we present a second example from a telecommunications transaction processing system. Consider the following specification of an account from a telephone service provider. The main purpose of the class is, simply, to charge the account whenever a phone-call finishes. The other operations of the class are, also, self-explanatory.

```
class Account
attributes
  object
  tel_number: Integer;
  balance: Integer:=0;
  charge_rate: Integer;
operations
  class
  *Create(client: Customer);
  object
  ?Balance(): Integer; // function, returns balance
  Charge(call_time: Integer);
body
methods
```

```
Charge
  is {
  set balance:= Balance()+call_time*charge_rate;
  }end
end class
```

A second class specification can be defined with the purpose of modelling the phone calls that each client makes. The operations specified here are used for illustrative purposes. Therefore, they are limited to the one that calculates the duration of a call and the one that determines the end of a call.

```
class Call
declarations
attributes
  object
  caller_number:Integer;
operations
  class
  *Create(client: Customer);
  object
  FinishCall();
  ?CalculateCallTime():Integer;
  body
methods
FinishCall
  is {
  // body of finish call detects end of call
  }end
CalculateCallTime
  is {
  // body-calculates the duration of the call
  }end
end class
```

In order to achieve the charging of the Account as soon as the phone-call ends we have to consider two possible scenarios, both related to the implementation of the two components. Either the Account and Call components are completely independent and are not aware of the existence of each other, in which case a third component is needed that becomes responsible for detecting the end of the phone-call, calculate the duration and perform the charge (Figure 2a), Or, Call is responsible for calling the Charge() method, for instance inside the FinishCall() method (Figure 2b). It should be clear that the latter would be a "weak" implementation. Indeed, it is hardly the role of a component that models phone-calls to charge an Account. However, such implementations often exist in real life applications. We argue that in the first scenario the best choice is to have a contract as the third component and that, in both cases, contracts provide a very effective way to evolve the system without modifying the existing components.

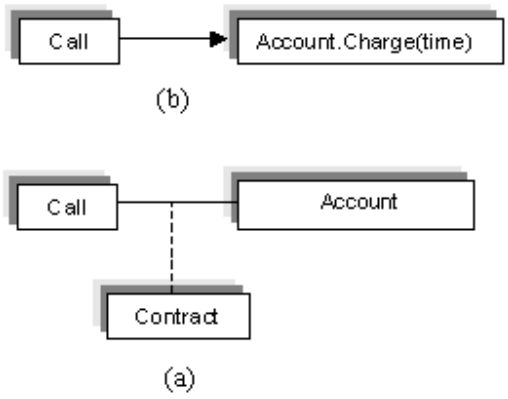

Figure 2. (a) Components Independent, (b) Components dependent

As far as the first scenario is concerned, the following simple contract, *Traditional Charging,* has the role of the third component and provides the required functionality while offering the advantage that the mechanism (contract) that controls the usage of the given objects is modelled as a first-class entity and, hence, can be evolved independently of the other two.

```
contract class Traditional Charging
participants x : Account; y : Call;
constraints x.tel_number:=y.caller_number;
coordination
  when *->>y.FinishCall();
  after
      local time: Integer:= y.CalculateCallTime();
      x.Charge(time);
end class
```

Consider now the situation in which the telephone provider wants to have two types of customers and charge them according to different rules. For instance, it could charge important customers only after the call exceeds a specific number of seconds, whereas not important customers are charged for the whole duration of their phone-call. If Account and Call are independent, i.e the first scenario described above is in place, the solution is simply to add to the system a contract like the one below.

```
contract class VIP_Charging
participants x : Account; y : Call;
attributes free_call_limit:Integer;
constraints x.tel_number:=y.caller_number;
coordination
  when *->>y.FinishCall();
  after
      local time: Integer:= y.CalculateCallTime();
      if (time> free_call_limit){
      x.Charge(time - free_call_limit);
```

```
    }
end class
```

The functionality of both the previous contracts is straightforward. They coordinate the charging of the Account according to the type of customer and the business rules the network operator defined. Notice that if a "*->>" is specified in the coordination part of the contract, any call to the service triggers the rule and that the keyword **local** just defines a local variable. If a future business requirement determines different behaviour for the components, a new contract, like the VIP_Charging contract above, can be added to the system in a "plug and play" mode in order to achieve the required behaviour.

Consider now the second scenario in which the two components, Account and Call, are aware of the existence of each other and that, in fact, an instance of a Call has to invoke the Charge() method in order to perform the charging of the customer's account (possibly as soon as the call ends). In this scenario, evolving the system to comply with the new requirement of having different charging schemes for different kinds of customers is not possible without modifying the components. For instance, consider the case in which inside FinishCall () there is a statement of the form Account.Charge (CalculateCallTime). Clearly, it is not possible to change the charging mechanism without changing the source code of either FinishCall() or Charge(). However, a contract like the one below can achieve the required functionality without having to modify the implementation of Call and Account.

```
contract class VIP_Charging_2
participants x : Account; y : Call;
attributes free_call_limit:Integer;
constraints x.tel_number:=y.caller_number;
coordination
  when y->> x.Charge(time);
  replace
     if (time > free_call_limit){
     local newtime: Integer:= time-free_call_limit;
     x.Charge(newtime);
     }
// implied "else" is void i.e. if time<free_call_time
// nothing is executed (it does not charge)
end class
```

A third scenario of evolution is the one in which we have different charging schemes related to the charge rate. For instance, a VIP Customer, can be charged with a charge_rate_1 when the duration of the call is within a time range [0-time_limit] and with a charge_rate_2 when the duration of the call exceeds time_limit. Again the following contract where the charging rates are decided inside the contract can offer a very effective and flexible solution.

```
contract class VIP_Charging_3
participants x : Account; y : Call;
attributes charge_rate_1, charge_rate2, time_limit :
Integer;
constraints x.tel_number:=y.caller_number;
coordination
  when *->>y.FinishCall();
  after
     local time: Integer:= y.CalculateCallTime();
     if (time <= time_limit){
  .  charge_rate:= charge_rate1;
     }
     else if(time > time_limit){
     x.charge_rate:= charge_rate2;
     }
     x.Charge(time);
end class
```

It should be noted that there are a large number of additional examples from different application domains that show how contracts can externalize the interactions between objects making them explicit in the conceptual model and support the compositional evolution of systems. Due to space limitations we will neither present such examples, nor the contracts formal semantics and the design pattern that puts them in practice. The reader can consult [1,2,3,4,7], for more details. In what follows we discuss how coordination contracts can be related to software tools.

3. Coordination Contracts and Tools

Coordination contracts can be related to tools in two ways: Firstly, in terms of tools that aim to apply coordination contracts as an intermediate stage in the development of larger systems and, secondly, in terms of using contract-based development for building software tools. The latter case draws from the realization that software tools are often themselves complex systems that are under constant evolution to cope with different requirements. As a result, techniques, such as coordination contracts, that aim to support software evolution in general, are also directly applicable to such tools. Therefore, it makes more sense to further discuss the former case only.

The former case is concerned with building tools to put in practice the concept of coordination contract. The implementation of such tools normally involves the following stages:

a. adopting the concepts of contracts to re-engineer components, in terms of making their functionality independent of their interconnection, thus

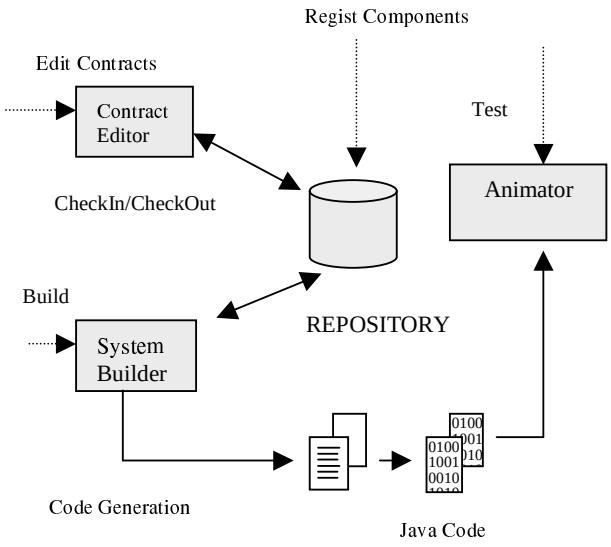

Figure 3. Coordination Contracts Tool Architecture

enabling externalization of volatile elements (business rules).

b. having *design/implementation techniques* that implement the contract concept in a way that satisfies the necessary requirements to achieve dynamic system evolution. The main technique is using a *Design Pattern* like the one presented in [3,7].

c. using a *specification tool* that will manage contract development and automated implementation of the pattern, by code generation/adaptation.

d. using a *configuration tool* that will "deploy", activating and deactivating, contracts.

Assuming implementation of the first stage, in other words, assuming having components of a suitable form either generated by a tool or coded by hand, we focus on the coordination layer of such tools. The activities of the development process that are supported by such tools are the following:

- *Registration*: components are registered in the tool.
- *Edition*: Contracts 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 some* parts according to the contract design pattern and *adaptation* of the given component part.
- *Animation*: some facilities are provided allowing testing/prototyping of contract semantics.

The logical architectural components, namely the Editor, the Repository, the Builder and the Animator that support the previous activities, are presented in Figure 3.

In this context, coordination contract tools may be applied to different levels in system development depending on several factors, such as the characteristics of the components, the way components are built, the development phase where the contract concept is going to be used, among others. We illustrate this diversity with two possible scenarios of using coordination contract tools:

Model Coordination: Coordination is used at the Analysis or Design phases. Components are model classes (e.g. UML classes) and 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 Implementation phase. Components are the final coded components of the basic building blocks of the system and coordination contracts are defined directly over implementation classes. It is suitable to be applied on the evolution of an existing system.

We realize, however, that the type of components to coordinate may define the context and capabilities in which such a tool is used and that the specific language and technical environment may impose constraints on the coordination features that can be used, since techniques to achieve the implementation of its semantics may not be available. We intend to further discuss and provide solutions for such issues in the future. However, to this end, we strongly believe that coordination contracts, its formal semantics that admits an implementation via design patterns and a contracts' development environment form a very strong basis for Software Engineers and developers to meet the challenge of designing and developing systems (and tools) that are better structured, consist of reusable parts, and are adaptive to change.

4. References

[1] L.F Andrade and J.Fiadeiro, “Evolution by Contract”, position paper presented at the *ECOOP'00 Workshop on Object-Oriented Architectural Evolution*.

[2] L.F.Andrade and J.L.Fiadeiro, "Interconnecting Objects via Contracts", *in UML'99 – Beyond the Standard*, R.France and B.Rumpe (eds), LNCS 1723, Springer Verlag 1999, 566-583.

[3] L.F.Andrade, J.L.Fiadeiro, J.Gouveia, A.Lopes and M.Wermelinger, "Patterns for Coordination", in *Coordination Languages and Models*, G.Catalin-Roman and A.Porto (eds), LNCS 1906, Springer-Verlag 2000, 317-322.

[4] L.Andrade and J.Fiadeiro, “Coordination: the evolutionary dimension”, in *Proc. TOOLS Europe 2001,* Prentice-Hall, in print.

[5] K.Chandy and J.Misra, Parallel Program Design - A Foundation, Addison-Wesley 1988.

[6] N.Francez and I.Forman, Interacting Processes, Addison-Wesley 1996.

[7] J.Gouveia, G.Koutsoukos, L.Andrade and J.Fiadeiro, “Tool Support for Coordination-Based Software Evolution”, *in Proc. TOOLS Europe 2001,* Prentice-Hall, in print.

[8] S.Katz, "A Superimposition Control Construct for Distributed Systems", *in ACM TOPLAS* 15, 1993 337-356

[9] B.Meyer, "Applying Design by Contract", *in IEEE Computer*, Oct.1992, 40-51.