

Managing Evolution in Telecommunication Systems

G.Koutsoukos*, J.Gouveia*, L.Andrade*, J.L. Fiadeiro**

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

***Department of Informatics
Faculty of Sciences, University of Lisbon
Campo Grande, 1700 Lisboa, PORTUGAL
jose@fiadeiro.org*

Keywords: *Component-based frameworks, Coordination, Evolution, Telecommunication Systems, Object Oriented Design, Reconfigurability, Scalability, Wireless Application Protocols*

Abstract: The recent advances in telecommunication technology, namely the wireless networks and the Internet, along with the competition of network operators for offering advanced and different services, are putting increasing pressure for building telecommunication software systems that are adaptive to new requirements and easily reconfigurable, even in run time. We show how a new modelling primitive – coordination contract – that we have developed and applied to other applications domains, can provide an effective solution to this problem. We describe coordination contracts and demonstrate, through several examples, how they can support the evolution of requirements of a telecommunications transaction processing system and of the specifications of the Wireless Application Protocol (WAP) Datagram layer. We also outline a development strategy based on coordination contracts that leads to systems that are more agile in reacting to change.

1 INTRODUCTION

Technology and system requirements in the telecommunications domain are changing very rapidly. Over the previous years, since the transition from analog to digital communications, and from wired to wireless networks, different standards and solutions have been adopted, implemented and modified, often to deal with new and different business requirements. More and more, telecommunication network operators strive to provide new advanced services in an attractive and usable way. However, time-to-market is a business decision that can be severely conditioned by the capacity of systems to accommodate changes quickly and with minimum impact on the services already implemented. This challenge is often difficult to be met by hardware-based systems because hardware cannot be easily modified and integrated.

On the other hand, thanks to the explosive growth of the Internet and the emergence of wireless data technologies, we are witnessing a major shift from hardware to software-based systems in this sector. This is because more and more applications must process data and information, a task that is easier to be performed on software. Therefore, it is not surprising that, due to their popularity in more traditional software application domains, object-oriented development techniques, such as the UML, and component-based frameworks like COM and CORBA, are becoming a standard in the telecommunications software industry.

However, it is now widely accepted that, although OO techniques such as inheritance and clientship make it easier to build systems, their support for evolution in general, and the ability of systems to exhibit the agility required by the volatility of business domains in particular, is quite limited. As explained in [1], [2], this is because interactions that take place in the application domain, corresponding to the services that the system is required to provide, are too often “hard-wired” in the code that implements the participating objects, making it hard to change or establish new interactions without having to change the implementation of the objects that model the more basic and stable entities of the domain as well. Yet, the ability to change is now much more important than the ability to create systems in the first place. Change has become a first-class design goal and requires both functional and technical architectures whose components can be added, modified, replaced and reconfigured dynamically.

In this paper, we argue that the modelling primitive – coordination contracts – that we presented in [1], [2], [3] for superposing coordination mechanisms over existing objects can be applied to telecommunication systems in order to achieve increased flexibility and agility in reacting to change. By borrowing concepts and techniques from Reconfigurable Distributed Systems and Software Architectures, coordination contracts provide the ability for interactions between objects to be modelled as first-class entities and for changes that require a reconfiguration of such interactions to be performed without having to change the objects involved. Through several examples related to the modelling of Wireless Application Protocols, we will give evidence on how coordination contracts can be used to support typical changes that are required by different business needs or imposed by the evolution of specifications.

2 COORDINATION CONTRACTS

In general terms, a coordination contract is a connection that is established between a group of objects (participants), where rules and constraints are superposed on the behaviour of the participants, which determines a specific form of interaction. The way such an 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 [5,7,10]. When a call is made from a client object to a supplier object, the contract “intercepts” the call and *superposes* whatever forms of behaviour 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 [2,4,9], allows coordination contracts to be superposed on given objects in a system to coordinate their behaviour without having to modify the way the objects are implemented (black box view).

Coordination contracts are currently supported by a specification language called Oblog [11] but the underlying technology is independent of the language and is being made available for other development platforms and environments. Using the Oblog notation, a coordination contract is defined as follows:

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

A contract declares a collection of participants that identify the classes of objects that can be partners in the contract, constraints that represent invariants defining the conditions under which instances from these classes may become partners in a contract instance, attributes and operations private to the contract, and the coordination effects that will be superposed on the partners to manage their interaction. 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 another applicable contract takes over or the original code of the trigger 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 contract’s own actions. When the trigger corresponds to the calling of an operation, three types of actions may be superposed on the execution of the operation:

1. **before** action: to be performed before the operation
2. **replace** action: to be performed instead of the operation (alternative)
3. **after** action: 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 different clauses is: all the “befores”, one “replace”, all the “afters”. Notice that the semantics of contracts allow for only one “replace” clause to be executed. The current implementation allows for a priority hierarchy to be specified among the alternatives. Any operation offered as an alternative in a “replace” clause is required to satisfy whatever properties have been specified on the original operation, e.g. in terms of pre/post-conditions.

The actions that are executed as part of the “do” clause are called the synchronisation set associated with the trigger. The semantics of contracts requires 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 and none of the actions in the synchronisation set is executed.

For a more detailed description of coordination contracts, the reader is urged to consult [1,2,3]: the presentation of the mathematical semantics that we have developed and the design pattern that we have used for implementing contracts are out of the scope of this paper. In the next sections we focus instead on illustrating the coordination role of contracts and present them as a means of structuring the evolution of telecommunication systems.

3 SUPPORTING EVOLVING REQUIREMENTS

Consider the following specification of an account from a telephone service provider (in a simplification of the full Oblog notation).

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

The main purpose of the class is, simply, to charge the account whenever a phone-call ends. The other operations of the class are, also, self-explanatory

A second class can be defined with the purpose of modelling the phone calls that each client makes. Because the operations specified here are used for illustrative purposes only, they are limited to the ones that calculate the duration and determine the end of a call.

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

In this context, how can the customer's account be charged as soon as the phone-call ends? There are two possible scenarios, both related to the implementation of the two components. Either the *Account* and *Call* components are 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; or the *Call* class is responsible for calling the *Charge()* method, for instance inside the *FinishCall()* method. It should be clear that the latter is a "weak" implementation. Indeed, it is hardly the role of a component that models phone-calls to charge an *Account*. However, such implementations are often the case in real life applications and we will take this possibility into account when showing how, in both cases, contracts provide a very effective way to evolve the system without modifying the existing components.

In the first scenario, the best design decision is to have the following contract in place of the mediating component:

```

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

```

The constraint specified on the participants ensures that the right account is being charged for the call. The coordination clause specifies the reaction to be performed when a call is finished: the duration of the call is determined and passed on to the account for the corresponding charge to be performed.

Having such a contract coordinating the way calls are charged instead of placing a direct invocation between the participating objects provides the functionality that is required while offering the advantage that the mechanism that controls the interaction between the given objects is modelled as a first-class entity and, hence, can be evolved independently of the other two. For instance, consider the situation

in which the telephone provider decides to distinguish between two types of customers and charge them according to different rules: VIP customers are charged only after the call exceeds a specific number of seconds, whereas other customers are charged for the whole duration of their phone-calls. In this scenario, all we need to accommodate this new business rule is for the contracts that regulate the way VIP customers are being charged to be replaced by an instance of the following new contract:

```
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(); // *-> : any call triggers the rule
  after
    local time: Integer:= y.CalculateCallTime(); // local: a local variable
    if (time> free_call_limit)
      x.Charge(time - free_call_limit);
end class
```

Through the use of the design pattern that we developed for deploying contracts [2,4,9], such a replacement can be performed without having to change the implementations of the components involved. Hence, when business requirements that are modelled through contracts change, the configuration of the system can be evolved accordingly in a “plug and play” mode, i.e. without intruding on the implementations of the components involved.

Notice that situations such as this one cannot be handled through conventional OO techniques, namely through the use of inheritance. Firstly, inheritance does not provide coordination as a first-class entity like contracts do, which means that interactions have to be “coded” directly in the components in terms of feature calling, and the implementation of the components to be changed to accommodate the new requirements. 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 the example above, this is the case when it is the type of client and not the type of account that determines the nature of the charges: it is not the accounts that are VIPs and, hence, changes to the way customers are charged should not lead to changes in accounts.

Consider now the second scenario in which the two components, *Account* and *Call*, are aware of the existence of each other in the sense that each 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

```

Through the implementation that we have developed for contracts [2,4,9], the previous scenario can be evolved to a more flexible solution that intercepts the direct interaction between calls and accounts and superposes the new contract.

Many other similar situations could have been used to illustrate how coordination contracts support evolution of requirements. For instance, different charging rates may be defined for different duration of calls. In such a case, contracts can offer a very flexible solution by deciding the charging rate and coordinating the charging procedure. Due to space limitations we will not present an example. However, we believe that the previous examples are enough to illustrate how contacts can externalise the interactions between objects, making them explicit in the conceptual model, and support compositional evolution of a telecommunication transaction processing system with respect to the evolution of business requirements.

4 THE WIRELESS DATAGRAM PROTOCOL

In this section, we make a more comprehensive account of the applicability of coordination contracts in the “wireless domain” by analysing the problem of supporting the evolution of port numbers and the evolution of wireless network bearer types and services in the implementation of the Wireless Application Protocol (WAP) Datagram Layer.

WAP is the latest attempt of the telecommunications industry to specify an application framework and network protocols for wireless devices with the main objective of bringing Internet content and advanced data services to digital cellular phones and other wireless terminals. A detailed description of the WAP architecture is presented in [12]. In general terms, the protocol layers are similar to the known OSI/ISO layers. In the context of this paper, the important aspect of the WAP architecture is that the WAP layers are designed to operate over a variety of different bearer services, supported by the various network types. This is accomplished in the layer referred to as the Wireless Datagram Protocol (WDP) [13].

WDP provides a common interface to the upper layers (Security, Session, and Application) so that they are able to function independently of the underlying wireless network. This is achieved by adapting the transport layer to specific

features of the underlying bearer. Therefore, the WAP layers architecture can, in fact, be considered as a 3 layered architecture of the upper layers, the underlying bearers and their interface (WDP). In general terms, WDP has to perform 3 tasks: port addressing by assigning port numbers (identify the higher layer entity above WDP), segmentation of datagrams and re-assembly of packets and error reporting. Discussing the way WDP performs these tasks is out of the scope of this paper. The reader can consult [13] for more details. Clearly, the list of supported bearers will change over time with new bearer types and services being added as the wireless market evolves (projection reached by WapForum in [12], pg 17). Moreover, specifications are changing in order to improve the protocol. As a consequence, relevant modifications to the implementation of the interface level (WDP) are needed in order to continue offering transparent services to the upper layers of the WAP stack. Therefore, WDP must be flexible enough to accommodate the changes in the underlying level quickly and with minimum impact on the services already implemented. In this section, we show how this flexibility can be achieved using the contract based development methodology. As far as the evolution of bearers is concerned, a generic architecture of our proposal is shown in Figure 1 below. In Figure 1, the WDP Components correspond to the operations of the WDP layer that are identical for all bearer services supported by WAP. This mainly means that they are computationally identical. However, their conditions for execution are different according to the underlying bearer. The WDP

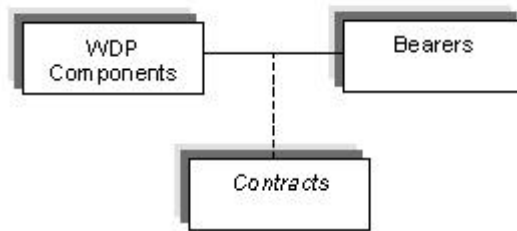


Figure 1. Contracts, bearers and WDP

Components in Figure 1 are components that can be implemented as “black boxes”. It is the responsibility of the contracts to coordinate the behaviour of such components according to the specific requirements of a bearer service. When a new bearer (type or service) is to be added to the ones already supported by WAP, new contracts will be added to the system to support that bearer. As a result, the already implemented WDP Components remain unchanged, thus allowing support for the evolution of requirements and achieving software reuse.

Consider now the case in which the evolution of WAP specifications results in new or even different specifications for WDP. For instance, the initial port numbers were different from the ones of later versions. Clearly, such modifications affect the already implemented parts of WDP that deal with port addressing. Consider the (real) case in which, in the implementation of the port addressing operation of a WAP Gateway (a part of the WAP network architecture that implements the

protocol), a component method called *wdp_udp_open()*, after receiving a WAP service, assigns a port number to its port attribute using, inside its body, statements of the following form (C notation):

```
if (strcmp(wap_service, "wsp") == 0) {
    port = 9200;
} else if (strcmp(wap_service, "wsp/wtp") == 0) {
    port = 9201;
// etc for all possible port numbers
} else {
    error(0,"Illegal configuration",wap_service);
    goto error; }
```

After performing the previous port number assignment – *wdp_udp_open()* – it calls a method – *udp_create_address(port)* – of another component, providing as argument the port number it has just assigned. Clearly, the previous code in which port number assignment is “hard-wired” inside the body of a method has to be rewritten in order to modify or include new port numbers. Still, if the code is rewritten following the above logic, the same problems will occur in the case a new specification for port numbers comes out. Therefore, from the evolution point of view, code such as the one above is undesirable.

In order to deal with these problems, a solution based on the use of contracts is proposed. Two objects, *Port_Address* and *UDP_Config* are the participants in a *Port_Assign* contract. *Port_Address* has as attributes a *wap_service* and a *port*, and a *wdp_udp_open()* operation that is supposed to provide the same functionality as the original real-life *wdp_udp_open()* operation presented above. Port number assignment is now performed in the coordination part of the contract. When the *udp_create_address(port)* is called, the port number passed as argument to the operation is defined in the *Port_Assign* contract. In that way, when new port numbers have to be introduced, a new contract will be plugged to the system to perform the updated port number assignment without affecting other parts of the code.

```
contract class Port_Assign
participants x: Port_Address , y: UDP_Config;
constraints x.wap_service!=NULL;
coordination
when x ->>(y.udp_create_address(x.port))
before{
    if x.wap_service == 'wsp';
    x.port=9200;
    //etc for all port numbers
end class
```

As already stated earlier, apart from port addressing, WDP has to provide segmentation and re-assembly of datagrams in a bearer dependent way. A datagram is a unit of information that consists of header fields and data fields. However, from the segmentation point of view, a datagram can be considered as a sequence of bits that is split into a number of packets being transmitted over the network. A reference number is used for distinguishing between different datagrams. Moreover, segmentation assigns headers to a packet containing the reference number for the WDP packet, the total number of segments in the datagram, and a segment number.

From the evolution point of view, the issue in segmentation is that the resulting packets must be of a size and format consistent with the underlying network service.

In a conventional design in which segmentation is implemented in different components in a bearer dependent way, the required evolution would be difficult to be achieved in a compositional way. However, contracts provide a very flexible solution to the problem. Consider a design (Figure 2) in which a class *Segmentation* defines a operation *Segment(Datagram)* to perform the segmentation of a datagram into a number of packets. The *Segmentation* class and *Segment* are defined in such a way that they provide the necessary computational functionality that is common for all bearer types. All bearer specific features of segmentation, such as packet size, encoding of packets and so on are modelled in contracts. Each contract corresponds to a bearer service and is responsible for coordinating the segmentation operation according to the underlying bearer requirements.

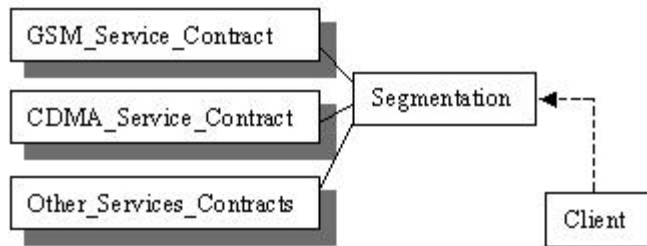


Figure 2. Contracts in Segmentation

For instance, *GSM_Service_Segmentation* could be the definition of a contract that is superposed on the *Segmentation* operation in order to support a GSM bearer service. The contract sets the maximum packet size for segmentation to be equal to the size required by the GSM Service. Moreover, the contract defines some operations for encoding the packet headers according to the particular GSM Service requirements. Naturally, additional operations or actions may be required based on more “low-level” design decisions.

```

contract class GSM_Service_Segmentation
constants gsm_N :Integer // number of bits per packet in the GSM service
participants x: Segmentation;
operations
    GSM_Service_Ref_Encod(int);
    // other operations
coordination
    when *->>x.Segment(Datagram) AND NETWORK.bearer_type:="GSM_Service";
    with Datagram.data !=NULL;
    before
        x.Size = gsm_N;
        ...
    replace
        ... // possibly replace the whole Segment() by an operation defined in a contract.

```

```

after
    for (int i=0, i<size (x.List_Packets), i++){
        x.List_Packets[i].ref_number=
            GSM_Service_Ref_Encod(x.List_Packets[i].ref_number);
    }
    // similarly for MaxFragment, Sequence_number
end class

```

When a new service is added to the protocol, a new contract will be added to the system in a “plug and play” mode to support that service. This allows for the *Segmentation* component already defined and all client components using *Segmentation* to remain unchanged. These features of not “breaking” the client and dynamically selecting alternatives of behaviour are some of the advantages of the contract-based design over the use of design patterns such as the Strategy or the Bridge [8] that could also be applied in order to provide a similar functionality. It is also important to note that the condition specified by the keyword AND under the “when” clause checks whether the network service is the correct one. This is accomplished by using a *NETWORK* global component, defined only for illustrative purposes. If the condition is false, it may be possible for another contract, imposed on *Segmentation* and concerning another network type, to be executed. In that way, support for dynamic network reconfiguration may be achieved. This is an application of contracts that we intend to further investigate in the future.

As far as the re-assembly case is concerned, the use of contracts for supporting the evolution of bearers is not so straightforward and, due to space limitations, we will not present an example. In what follows, we briefly discuss how contracts can be applied to support the error reporting mechanism of WAP.

Processing errors can happen when WDP datagrams are sent from a WDP provider to another. For instance, there may be no application listening to the destination port, or the receiver may not have enough buffer space to receive a large message. Routines report such errors by generating error messages of a specific format. Discussing the error reporting mechanism of WAP is out of the scope of this paper. The reader can consult [14] for more details. The relationship between the error messages’ format and the underlying bearer is specified in the **Address Information field**. The address information field consists of Address Type, Address Length and Address Data elements. Address Type specifies the type of the underlying bearer. If the Address Type is GSM, the Address Data must be coded using the semi-octet representation defined in GSM 03.40 [6]. Similar requirements are specified in [14] for all the different bearers supported by WAP. This indicates that different bearer types have different message formats. Building the Information field format inside the routine in a bearer dependent way would lead to interactions that are “hard-wired” in the code making it difficult to introduce changes related to the evolution of bearers. Therefore, in order to support the evolution of bearers, these elements of Error messages have to be generated in a bearer independent way. The following example illustrates how contracts provide a convenient way to achieve this.

A class of objects, *Err_Msg*, may be defined to model the Error Messages and their attributes, according to the WCMP specification [14]. A class *Address_Info*, is responsible for generating, using a *generate_Address_Info()* operation, the

Address_Information fields i.e *Addr_type*, *Addr_Length*, *Addr_Data*, and assigning their values to the corresponding attributes of *Err_Msg*. In order for having these fields generated in a bearer independent way, a contract, *GSM_Addr_Information*, is defined. The contract is related to the GSM bearer type case. A similar contract would be defined for any other wireless network. The role of the contract(s) is to generate the Address Information of messages according to the bearer requirements.

```

contract class GSM_Addr_Information
participants x: Address_Info;
operations
class
    GSM_0340_Encod(Datagram); //methods to perform Encoding according to GSM 03.40
    GSM_Addr_Length_Encod(Datagram);
coordination
    when *->x.generate_Address_Info(Err_Msg, Datagram) AND NETWORK.bearer_type=="GSM";
    before
        x.Addr_type := "GSM";
        x.Addr_Length := GSM_Addr_Length_Encod(Datagram);
        x.Addr_Data := GSM_0340_Encod(Datagram);
end class

```

Generate_Address_Info provides a uniform interface for all routines in the system. A routine, after encountering an error, calls *generate_Address_Info* to build the Address Information Field of the message. How this field will be built is co-ordinated by the contract according to the underlying bearer. The contract intercepts the call and ensures that the Address Information field will be built according to bearer requirements. As a result, when new bearer types are added to WAP, new contracts will be added to support them without affecting the functionality of the rest of the system. Moreover, as explained in the segmentation case, contract-based designs provide a number of advantages over existing design patterns. Due to space limitations we do not discuss such issues in more detail. We intend, however, to do so in future papers.

5 CONCLUDING REMARKS

The telecommunications sector is being governed by the expeditious growth of two networking technologies: the wireless data and the Internet. This growth has fuelled the creation of new and exciting information services and resulted in a major shift from hardware to software as far as the implementation of telecommunication systems is concerned. However, under this increasing pressure for new sophisticated services, and with the frequent adoption of new standards, software development in telecommunications cannot rely solely on traditional OO development techniques such as inheritance and clientship. We believe that the only hope for telecommunication organisations to be able to face the challenges of the fast market and technology evolution is to follow the emerging trend in software analysis and design based on predefined frameworks of skeletal applications, components, and design patterns that can be easily customized and integrated.

In this paper, we proposed the contract-based development methodology described in [1], [2], [3] as a discipline that can be applied when developing telecommunications systems in order to support the evolution of system requirements. We supported our view by presenting examples in which contracts provide a compositional structuring mechanism with respect to changes occurring due to the evolution of requirements in a telecommunications transaction-processing system. We also showed that contracts can be applied to the Port addressing, Segmentation and Error reporting operations of the Wireless Application Protocol Datagram Layer in order to accommodate changes imposed by the evolution of the protocol's specifications. We did not attempt, however, to provide "low level" designs and, therefore, our examples lack implementation details. Based on our experience in applying the contract-based methodology in other domains, we believe that, by refining these designs to real implementations, increased levels of flexibility and reuse will be achieved.

We are also certain that these methods can be applied to other areas in the telecommunications domain. A good basis for meeting this opportunity can be provided by adopting a development strategy based on the following steps:

1. Decompose the system into parts (the WAP already consists of multiple layers but further decomposition into smaller parts is possible).
2. Determine evolution critical parts (the evolution of bearers in WAP implies that WDP is an evolution critical part).
3. Obtain a thorough understanding of the domain of the specific problem being solved.
4. Model individual components as domain objects and identify operations that are "stable" (computationally identical).
5. Coordinate the joint behaviour of the previous objects through the superposition of contracts.

Clearly, additional issues need to be addressed before these steps can be effectively applied. For instance, one could ask how we can determine the evolution critical parts, and how we can identify the "stable" operations in a system. Naturally, these questions can only be answered by experts in the problem domain. We are currently investigating criteria that would assist such experts in providing answers to the previous questions. Moreover, because performance is a critical concern for telecommunication applications, we are also evaluating the impact that the design pattern, presented in [2], [4], [9], as a way to implement contracts, has on the performance of systems.

6 ACKNOWLEDGEMENTS

Some of the ideas presented in this paper concerning the use of coordination contracts in telecommunications were developed as part of the first author's MSc thesis at King's College, University of London, September 2000. The first author wishes to acknowledge Prof. Tom Maibaum for his valuable guidance, comments and moral support

7 REFERENCES

- [1] L.F Andrade and J.L 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 , pp. 566-583,1999.
- [3] L.F Andrade and J. L Fiadeiro. Coordination: the evolutionary dimension. in *Proc. TOOLS Europe 2001*, Prentice-Hall, in print.
- [4] L.Andrade, J.Fiadeiro, J.Gouveia, A.Lopes and M.Wermelinger. Patterns for Coordination. In *Coodination Languages and Models*, G.Catalin-Roman and A.Porto (eds), LNCS 1906, pp. 317-322, Springer-Verlag 2000.
- [5] K.Chandy and J.Misra. *Parallel Program Design – A Foundation*, Addison-Wesley, 1988.
- [6] ETSI (European Telecommunication Standardisation Institute) European Digital Cellular Telecommunication Systems (phase 2+). *Technical realisation of the Short Message Service (SMS) Point-to-Point (P)* (GSM 03.40 version 7.1.0 Release 1998.
- [7] N.Francez and I.Forman. *Interacting Processes*, Addison-Wesley 1996.
- [8] E. Gamma, R.Helm, R. Johnson and J. Vlissidis. *Design Patterns: Elements of Reusable Object Oriented Software*, Addison Wesley, 1995.
- [9] J.Gouveia, G.Koutsoukos, L.Andrade and J.Fiadeiro. Tool support for coordination based evolution. In *Proc. TOOLS Europe 2001*, Prentice-Hall, in print.
- [10] S.Katz. A Superimposition Control Construct for Distributed Systems. In *ACM TOPLAS 15(2)*, 1993, 337-356.
- [11] The Oblog Corporation. “The Oblog Specification Language”, <http://www.oblog.com/tech/spec.html>.
- [12] The WapForum. “The WAP Architecture Specification”,Version 30th April 1998 <http://www.wapforum.org/what/technical.htm>.
- [13] The WapForum. “The WAP Wireless Datagram Protocol Specification”, Version 19th Feb 2000, <http://www.wapforum.org/what/technical.htm>.
- [14] The WapForum. “TheWireless Control Message Protocol”, Version 19th Feb. 2000 <http://www.wapforum.org/what/technical.htm>.