The Coordination Development Environment

João Gouveia[1], Georgios Koutsoukos[1], Michel Wermelinger[2,3], Luís Andrade[1,3], and José Luiz Fiadeiro[3,4]

[1] Oblog Software SA
Alameda António Sérgio 7, 1A, 2795-023 Linda-a-Velha, Portugal
{jgouveia,gkoutsoukos}@oblog.pt
[2] Dep. de Informática, Fac. de Ciências e Tecnologia, Univ. Nova de Lisboa
2829-516 Caparica, Portugal
http://ctp.di.fct.unl.pt/~mw
[3] ATX Software SA
Alameda António Sérgio 7, 1C, 2795-023 Linda-a-Velha, Portugal
landrade@atxsoftware.com
http://www.atxsoftware.com
[4] Dep. de Informática, Fac. de Ciências, Univ. de Lisboa
Campo Grande, 1700 Lisboa, Portugal
http://www.fiadeiro.org/jose

1 The Concept

Coordination contracts [1,2] are a modelling primitive, based on methodological and mathematical principles [8,3], that facilitates the evolution of software systems. The use of coordination contracts encourages the separation of computation from coordination aspects, and the analysis of which are the "stable" and "unstable" entities of the system regarding evolution. Coordination contracts encapsulate the coordination aspects, i.e., the way components interact, and as such may capture the business rules [7] or the protocols [6] that govern interactions within the application and between the application and its environment.

System evolution consists in adding and removing contracts (i.e., changing the business rules) between given components (the participants of the contract). As a result of an addition, the interactions specified by the contract are superposed on the functionalities provided by the participants without having to modify the computations that implement them. In fact, the components are completely unaware they are being coordinated. The contracts specify a set of rules, each with a triggering condition (e.g., a call to a method of one participant), and a rule body stating what to do in that case. Contracts are also unaware of the existence of other contracts. This facilitates enormously incremental system evolution, because explicit dependencies between the different parts of the system are kept to a minimum, and new contracts can be defined and added to the system at any time (even run-time), thus coping with changes that were not predicted at system design time.

Consider the banking domain, in which ATX Software has several years of experience. Usually, there is an object class account 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 [2], assigning this guard to withdrawal can be seen as part of the specification of a business requirement and not necessarily of the functionality of a basic business entity like account. Indeed, the circumstances under which a withdrawal will be accepted can change from customer to customer and, even for the same customer, from one account to another depending on its type.

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

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

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

The constraint means that instances of this contract can only be applied to instances of Customer that own the corresponding instance of Account. The coordination rule is only triggered when the participating Customer calls the withdrawal operation of the participating Account. The rule superposes the guard (after the `with` keyword) that restricts withdrawals to states in which the balance is greater than the requested amount. If the guard is false, the rule fails, i.e., the withdrawal operation is not executed.

Having externalized the "business rule" that determines the conditions under which withdrawals can be made, we can support its evolution by defining and superposing new contracts. For instance, consider a contract that a customer may subscribe to instead of standard-withdrawal: whenever the balance is less than the amount, instead of occurring a failure, the whole balance would be withdrawn.

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

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

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

2 The Tool

For this approach to be usable in real applications, it requires a tool to support system development and evolution using coordination contracts. Capitalising on the expertise of Oblog Software in building development tools, the Coordination Development Environment (CDE) we envisage [5] allows the following activities:

Registration: component types (developed separately) are registered to the tool as candidates for coordination.

Edition: contract types are defined, with participants taken from the available component types.

Deployment: the code necessary to implement the coordinated components and the contract semantics is generated. This code is then compiled and linked (ouside of CDE) with the non-coordinated components to produce the complete application.

Configuration: contracts (i.e., instances of contract types) are created or removed between given components (i.e., instances of component types) at run-time and the values of the attributes can be changed.

Animation: the run-time behaviour of contracts and their participants can be observed, to allow testing of the application.

The current version of CDE helps programmers to develop Java applications using coordination contracts. More precisely, it allows to write contracts (in a concrete syntax different from the modelling syntax of the previous section), to translate them into Java, and to register Java classes (components) for coordination. The code for adapting those components and for implementing the contract semantics is generated based on a micro-architecture we developed [5] that is based on the Proxy and Chain of Responsibility design patterns. The CDE also includes an animation tool, with some reconfiguration capabilities, in which the run-time behaviour of contracts and their participants can be observed using sequence diagrams, thus allowing testing of the deployed application. Future work will include the implementation of coordination contexts [4], a modelling primitive to specify reconfiguration actions.

CDE is written in Java and requires JDK 1.2. Its first public release is freely available for download from the ATX website (`http://www.atxsoftware.com`).

References

1. L. Andrade and J. L. Fiadeiro. Interconnecting objects via contracts. In *UML'99 -Beyond the Standard*, LNCS 1723, pp. 566–583. Springer-Verlag, 1999.
2. L. Andrade and J. L. Fiadeiro. Coordination technologies for managing information system evolution. In *Proc. CAiSE'01*, LNCS 2068, pp. 374–387. Springer-Verlag, 2001.
3. L. Andrade and J. L. Fiadeiro. Coordination: the evolutionary dimension. In *Proc. TOOLS 38*, pp. 136–147. IEEE Computer Society Press, 2001.
4. L. Andrade, J. L. Fiadeiro, and M. Wermelinger. Enforcing business policies through automated reconfiguration. In *Proc. of the 16th IEEE Intl. Conf. on Automated Software Engineering*, pp. 426–429. IEEE Computer Society Press, 2001.
5. J. Gouveia, G. Koutsoukos, L. Andrade, and J. L. Fiadeiro. Tool support for coordination-based software evolution. In *Proc. TOOLS 38*, pp. 184–196. IEEE Computer Society Press, 2001.
6. G. Koutsoukos, J. Gouveia, L. Andrade, and J. L. Fiadeiro. Managing evolution in telecommunication systems. In *New Developments in Distributed Applications and Interoperable Systems*, pp. 133–139. Kluwer, 2001.
7. 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 *Proc. of the ECOOP Workshop on Object Oriented Business Solutions*, pp. 41–52, 2001. Invited paper.
8. A. Lopes and J. L. Fiadeiro. Using explicit state to describe architectures. In *Proc. of Fundamental Approaches to Software Engineering*, LNCS 1577, pp. 144–160. Springer-Verlag, 1999.