A Coordination Methodology and Technology for Agile Businesses

Luís Andrade[1], José Luiz Fiadeiro[1,2],
João Gouveia[1], Georgios Koutsoukos[1], Michel Wermelinger[1,3]
[1]ATX Software SA, Alameda António Sérgio 7, 1C, 2795-023 Linda-a-Velha, Portugal
[2]Dep. de Informática, Fac. de Ciências, Univ. de Lisboa, Campo Grande, 1700 Lisboa, Portugal
[3]Dep. de Informática, Univ. Nova de Lisboa, 2829-516 Caparica, Portugal
{landrade, jgouveia, gkoutsoukos}@atxsoftware.com
jose@fiadeiro.org mw@di.fct.unl.pt

1. MOTIVATION

The engineering of Business Systems is under the increasing pressure to come up with software solutions that allow companies to face very volatile and turbulent environments (as in the telecommunications domain [3]). This means that the complexity of software has definitely shifted from construction to evolution, and that new methods and technologies are required.

Most often, the nature of changes that occur in the business are not at the level of the components that model business entities, but at the level of the business rules that regulate the interactions between the entities. Therefore, we believe that sucessful methodologies and technologies will have to provide abstractions that reflect the architecture of such systems by supporting a clear separation between computation, as performed by the core business components, and coordination, as prescribed by business rules. This separation should help in localising change in the system, both in terms of identifying what needs to be changed in the system and circumscribing the effects of those changes.

In our opinion, the lack of abstractions for supporting the modelling of interactions and architectures explains why component-based and object-oriented approaches have not been able to deliver their full promise regarding system evolution. Usually, interactions are coded in the way messages are passed, features are called, and objects are composed, leading to intricate spaghetti-like structures that are difficult to understand, let alone change. Moreover, new behaviour is often introduced through new subclasses which do not derive from the "logic" of the business domain, widening the gap between specification and design.

The approach we have been developing [1] builds on previous work on coordination models and languages, software architecture, and parallel program design languages. Instead of delegation we use explicit architectural connectors that encapsulate coordination aspects: this makes a clear separation between computations and interactions and externalises the architecture of the system. Instead of subclassing we advocate superposition as a structuring principle: interactions are superposed on components in a non-intrusive and incremental way, allowing evolution through reconfiguration, even at run-time.

The main advantages of our approach are adequacy and flexibility. The former is achieved by having a strict separation of computation, coordination, and configuration, with one primitive for each concept, stating clearly the pre-conditions for each coordination and reconfiguration rule. As for flexibility, interactions among components can be easily altered at run-time through (un)plugging of coordination rules, and it is possible to state exactly which coordination rules are in effect for which components, and which configuration policies apply to which parts of the system.

In the following sections we briefly summarise our approach for different phases of software development. More details are provided by the publications available at www.atxsoftware.com.

2. LAWS FOR DESIGN

At the modelling level, business entities and rules are specified in an implementation-independent way, by *coordination interfaces* and *coordination laws*, respectively. A coordination interface indicates a collection of services that must be provided and events that must be generated by all components that implement the interface. A coordination law states when and how to react to the events of the components that are subject to that law. Put in another way, services are provisions, events are requests, and laws bind provisions to requests, stating exactly when a request is attended and which services are used to process it. We stress that services and events are at the conceptual level; the implementation artefacts (messages, remote procedure calls, interrupts, etc.) they are mapped to depend on the implementation platform.

As an example, taken from the banking domain, consider two kinds of components: customers and accounts. We wish to model two business rules: normal customers may not overdraw their accounts, but VIP customers may overdraw up to some limit which is negotiated between the bank and the customer. For these rules, the full specification of accounts and customers is irrelevant. The only necessary services and events are given by the following interfaces:

coordination interface account-debit
type-id account
services debit(m: money); balance() : money
properties balance() **after** debit(m) **is** balance() - m
end interface

coordination interface customer-withdrawal

OOPSLA'02, November 4-8, 2002, Seattle, Washington, USA.
ACM 1-58113-626-9/02/0011.

type-id customer
services owns(a: account) : boolean
events withdraw(m: money; a: account)
end interface

Notice that there is no pre-condition on the 'debit' operation of the account because such interaction rules may be subject to evolution and hence are best given by coordination laws. The law for normal customers is as follows.

coordination law normal-withdrawal
partners a:account-debit, c:customer-withdrawal
when c.withdraw(m,a)
with a.balance()$\geq$ n **and** c.owns(a)
do a.debit(n);
end law

As can be seen, each law has a set of subjects, indicated by the interfaces they must implement, and a coordination rule. The rule consists of an event (**when**), a guard (**with**) and a body (**do**). Whenever the event occurs, if the guard is true then the rule body is executed; if the guard is false, the body is not executed and a failure is reported.

The law for VIP customers has a local attribute to hold the limit up to which the customer may overdraw the account.

coordination law vip-withdrawal
partners a:account-debit, c:customer-withdrawal
attributes credit:money
when c.withdraw(m,a)
with a.balance() + credit $\geq$ n **and** c.owns(a)
do a.debit(n);
end law

It is important to notice that laws are instantiated to groups of component instances, which allows complete flexibility in setting up which business rules govern which interactions among business entities, and how. For the example at hand, it is possible to have an account owned by three customers such that one customer cannot overdraw the account, and the other two can overdraw it with different limits.

Last, but not least, the use of superposition means that the combination of laws has a cumulative effect. If the same event triggers two or more laws, the union of the bodies is executed atomically if the conjunction of the guards is true, and jointly fails if any of the guards is false.

3. CONTRACTS FOR IMPLEMENTATION

Coordination laws can be instantiated for specific types of components, events and services once the target development platform has been chosen. This instantiation leads to what we call *coordination contracts*, which correspond to connector types as known in the area of Software Architectures. Coordination contracts are then applied to specific component instances that implement the prescribed coordination interfaces.

In the environment that we built for supporting the proposed coordination-based development method, component types are Java classes and events consist, basically, of method calls.This means that for laws to be properly enforced, contracts must intercept in a transparent way the messages sent among contract participants. Transparent means that components are not aware they are being coordinated, and therefore have no way to bypass the contracts. For this to be achieved, a design pattern [2] was developed that allows contracts to be used without programmers having to modify the source code of the application classes. The tool automatically generates the design pattern implementation for the components indicated to be under coordination.

Due to space limitations, for coordination contract examples we refer the readers to the listed papers.

4. CONTEXTS FOR CONFIGURATION

A running system consists of a set of components and a set of contracts establishing interconnections between components. The configuration of the system can evolve—in order to reflect how the system adapts itself or reacts to changes in business context—by removing, adding, and replacing components and contracts. However, such changes do not occur in an ad-hoc fashion. Normally, there are restrictions on which laws can be applied to which components, and to which changes are possible.

Coordination contexts are the primitive we propose to capture business activities that include not only access to the services provided by the system but also reconfiguration operations. A context provides a "view", or "gateway", through which an agent (be it human or computational) can interact with the system. Each context is available to only some agents. For instance, the following context allows to manage the accounts of a given customer, and can only be available to a bank manager, because customers may not change their normal/VIP status.

coordination context AccountManagement (c : customer)
  component types Account, Customer
  contract types Normal, VIP
  constants maxCredit: money = 100000
  invariants forall a:Account c.owns(a) **implies**
      exists Normal(c,a) **xor exists** VIP(c,a)
  configuration services
    new_account(a:account, m:money):
      pre not exists a
      post exists a **and** c.owns(a) **and exists** Normal(c,a) **and**
        a.balance() = m
    subscribe_VIP(a:account,limit:money):
      pre exists Normal(c,a) **and** limit $\leq$ maxCredit
      post exists VIP(c,a) **and** VIP(c,a).credit = limit **and**
        not exists Normal(c,a)
  configuration rules
    automaticVIP:
      when exists Normal(c,a) **and** a.balance() > 100000
      post exists VIP(c,a) **and** VIP(c,a).credit=10000 **and**
        not exists Normal(c,a)

As can be seen from the example, contexts provide both for services to be invoked on demand by the agents as well as rules that are to be automatically applied when some conditions become true. This helps in maintaining invariants over the part of the system that belongs to the context.

5. REFERENCES

[1] L. Andrade and J. L. Fiadeiro. Coordination: the evolutionary dimension. In *Proc. TOOLS 38*, pages 136–147. IEEE Computer Society Press, 2001.

[2] J. Gouveia, G. Koutsoukos, L. Andrade, and J. L. Fiadeiro. Tool support for coordination-based software evolution. In *Proc. TOOLS 38*, pages 184–196. IEEE Computer Society Press, 2001.

[3] G. Koutsoukos, J. Gouveia, L. Andrade, and J. L. Fiadeiro. Managing evolution in telecommunication systems. In *New Developments in Distributed Applications and Interoperable Systems*, pages 133–139. Kluwer, 2001.