Patterns for Coordination

L.F.Andrade[1], J.L.Fiadeiro[2,3*], J.Gouveia[1], A.Lopes[3] and M.Wermelinger[4]

[1] OBLOG Software S.A., Alameda António Sérgio 7 – 1 A,
2795 Linda-a-Velha, Portugal
{landrade,jgouveia}@oblog.pt

[2] Department of Computer Science, King's College London
Strand, London WC2R 2LS, UK
jose@fiadeiro.org

[3] Department of Informatics, Faculty of Sciences, University of Lisbon
Campo Grande, 1700 Lisboa, Portugal
mal@di.fc.ul.pt

[4] Dep. of Informatics, Fac. Sciences and Technology, New University of Lisbon
Quinta da Torre, 2825 Monte da Caparica, Portugal
mw@di.fct.unl.pt

1 Introduction

The separation between computation and coordination, as proposed by recent languages and models [7], has opened important new perspectives for supporting extendibility of systems, i.e. the possibility of adapting software systems to changes in requirements in an easy way. The evolutionary model that we have been developing is based on the representation of the more volatile aspects of the application domain like business rules as connectors whose purpose is to coordinate the interaction among core, more stable, components. The idea is that, in this way, evolution can be made to be compositional over the changes that occur in the application domain through the addition, deletion or substitution of connectors, without interfering with the services provided by the core objects of the system.

The applicability of this particular approach to evolution has been demonstrated in the field through the development of software systems in a number of domains using the OBLOG tool [www.oblog.com]. OBLOG provides a language and family of tools for supporting object-oriented development. One of the features that distinguish it from other products in the market is its collection of primitives for modelling the behavioural aspects of systems. These include the notion of *contract* [2], a semantic primitive corresponding to connectors as used in Software Architectures [1], which we brought into OO modelling as a means of supporting the coordination layer required by the approach that we described above.

Ideally, this method for evolving systems should be applicable regardless of the languages in which the core objects are programmed, thus allowing for the integration of third-party, closed components, like legacy systems. For such a general support to

[*] On leave at (2) from (3) with the support of *Fundação para a Ciência e Tecnologia* and the *EPSRC*.

be possible, we need to abstract away from the specific coordination languages and models that have been proposed in the literature, universal principles that can be incorporated into a tool like OBLOG.

Our purpose in this paper is to relate progress that we have made so far into that endeavour. In section 2, we present a "mathematical pattern" that identifies essential mechanisms of coordination present in languages for parallel program design and architecture description. This pattern was used in [2] for giving semantics to contracts. In section 3, we present a design pattern that allows for such mechanisms to be implemented in platforms for component-based system development like CORBA, EJB and COM. This pattern was used for implementing contracts in OBLOG.

2 A Categorical Pattern for Coordination

Our mathematical approach to coordination was first outlined in [4]. It is based on the use of Category Theory as a means of formalising complex configurations of interconnected components, an approach that can be traced back to Goguen's work on General Systems Theory. The basic motto of the categorical approach to systems is that morphisms can be used to express interaction between components, so that "given a category of widgets, the operation of putting a system of widgets together to form some super-widget corresponds to taking the colimit of the diagram of widgets that shows how to interconnect them" [5].

The specific application of this approach to coordination abstracts general principles from previous applications of categorical techniques to parallel program design centred on the notion of superposition[8]. In this approach, we assume that a notion of system (be it system specifications, models or designs) can be organised in a category ***SYS*** whose morphisms capture the relationship that exists between systems and their components (e.g. superposition). In this setting, we take the separation between coordination and computation to be materialised through a functor $\boldsymbol{int}:\boldsymbol{SYS}\rightarrow\boldsymbol{INT}$ mapping the category of systems to a category of interfaces that represent the elements through which interconnections between system components can be established.

The fact that ***int*** "forgets" the computational aspects of systems, and that these do not play any role in the interconnections, is captured by the following properties:

- ***int*** *is faithful;*
- ***int*** *lifts colimits;*
- ***int*** *has discrete structures.*

The fact that ***int*** is faithful (injective over each hom-set), means that morphisms of systems cannot induce more relationships between systems than between their underlying interfaces. That is, by taking into consideration the computational part, we do not get additional observational power over the external behaviour of systems.

The second property means that, given any configuration diagram $\boldsymbol{dia}:\boldsymbol{I}\rightarrow\boldsymbol{SYS}$ over systems and colimit $(\boldsymbol{int}(S_i)\rightarrow C)_{i:\boldsymbol{I}}$ of $(\boldsymbol{dia};\boldsymbol{int})$, there exists a colimit $(S_i\rightarrow S)_{i:\boldsymbol{I}}$ of ***dia*** such that $\boldsymbol{int}(S_i\rightarrow S)=(\boldsymbol{int}(S_i)\rightarrow C)$. In other words, if we interconnect system components through a diagram, then any colimit of the underlying diagram of interfaces establishes an interface for which a computational part exists that captures the joint behaviour of the interconnected systems. This is a "correctness" result in the sense that it states that, by interconnecting the interfaces of systems, we are interconnecting the systems themselves.

The corresponding "completeness" result – that all interconnections can be established via interfaces – is given by the third property. The fact that $\boldsymbol{int}$ has discrete structures means that, for every interface $C{:}\boldsymbol{INT}$ there exists a system $s(C){:}\boldsymbol{SYS}$ such that, for every interface morphism $f{:}C\to\boldsymbol{int}(S)$, there is a system morphism $g{:}s(C)\to S$ such that $\boldsymbol{int}(g)=f$. That is to say, every interface C has a “realisation” (a discrete lift) as a system $s(C)$ in the sense that, using C to interconnect a component S, which is achieved through a morphism $f{:}C\to\boldsymbol{int}(S)$, is tantamount to using $s(C)$ through any $g{:}s(C)\to S$ such that $\boldsymbol{int}(g)=f$.

In this setting, interconnections are usually expressed through diagrams of the form

$$\begin{tikzcd}
 & \text{channel}_1 \arrow[dl] \arrow[dr] & & \text{channel}_2 \arrow[dl] \arrow[dr] & \\
\text{comp}_1 & & \text{regulator} & & \text{comp}_2
\end{tikzcd}$$

where the channels are of the form $s(C)$ for some interface C and the regulator is a component that superposes the coordination mechanisms that are required for regulating the interaction between the two components. See [2,3] for examples.

3 A Design Pattern for Coordination

The fact that a mathematical pattern exists for justifying our principle of supporting evolution through the superposition of connectors does not mean that it can be put into practice directly over the technology that is available today. In this section we show that, even if none of the standards for component-based software development that have emerged in the last few years (e.g. CORBA, EJB and COM) can provide a convenient and abstract way of supporting superposition as a first-class mechanism, an implementation can be given that is based on a design pattern that exploits some widely available properties of object-oriented programming languages such as polymorphism and subtyping.

Before we discuss the design pattern that we developed for coordination, we must point our that the level at which its "correctness" with respect to the mathematical pattern discussed in the previous section can be established is not a mathematical one. This is because neither the mathematical semantics nor mathematical abstractions of these platforms for component-based system development are available. Hence, we will argue for, rather than prove, its correctness. Furthermore, we did not take it as a task to "implement" the notions of object, morphism, colimit, etc. Instead, we have tried to make sure that the "spirit" of the mathematical solution was captured by the implementation pattern: we provide autonomous existence to interfaces, use these as a means for interconnecting programs, and ensure that interconnections are not lost by restricting them to the interfaces.

The class diagram below depicts the proposed pattern, based on well-known design patterns, namely the Proxy or Surrogate [6]. Its "correctness" relies on two main mechanisms. On the one hand, provision of a specific interface *(SubjectInterface)*, as an abstract class, for every component. This interface is linked to the real program *(SubjectBody)* through a dynamically reconfigurable proxy reference. On the other hand, support for dynamic reconfiguration of the code executed upon requests for operations (including requests by *self* as in active objects), 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 (*ImplementationProxy*) – that complies with the interface of the component. On the other hand, the type of its values may vary at run-time through *Channel* 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 (*SubjectBody* or *Channel*) based on the type to which the proxy belongs. Relying on this mechanism of later binding, the reconfiguration by superposition is implemented by (1) introducing the intended coordinated behaviour of the parties, as obtained through the colimit, on *Regulator* objects, and (2) connecting them to the parties using the *Channel* objects as explained below.

In what follows, we explain, in more detail, the basic features of the pattern, starting with the participating classes.

SubjectInterface – as the name indicates, this is an abstract class (type) that defines the common interface of services provided by *ImplementationProxy* and *Subject*.

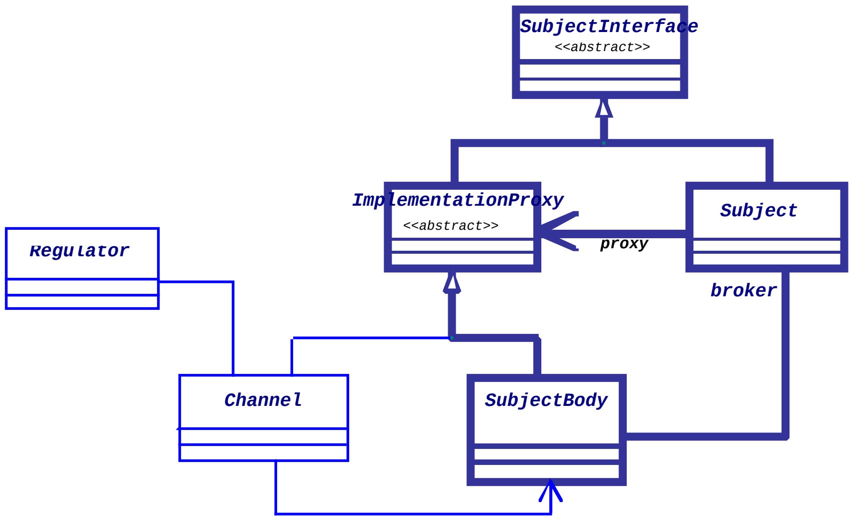
Subject – This is a concrete class that implements a broker maintaining a reference that lets the subject delegate received requests to the abstract implementation (*ImplementationProxy*) using the polymorphic entity proxy. At run-time, this entity may point to a *SubjectBody* if no regulator is active, or to the *Channel* that links the real subject to the regulators that coordinate its behaviour.

ImplementationProxy – This is an abstract class that defines the common interface of *SubjectBody* and *Channel*. The interface is inherited from *SubjectInterface* to guarantee that all these classes offer the same interface as *Subject* (the broker) with which clients of the real subject have to interact.

SubjectBody – This is 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.

Channel – This class maintains the interconnections between the real object (*SubjectBody*) and other components of the system. Adding or removing interconnections with the same real object does not require the creation of a new instance of this class but only of a new association with the new regulator and an instantiation link to the existing instance of *Channel*. This means that there is only one instance of this class associated with one instance of *SubjectBody*.

Regulator – This is an object that subsumes the coordination specified through the configuration diagrams defined in the previous section (colimit semantics); it is notified and takes decisions whenever a request is invoked on a real subject.



A typical execution of the pattern starts with a request for an operation of the object. Because clients interact with the real object via the broker, the call to any operation of the real object is handled by the broker *(Subject)*. The broker then uses the polymorphic entity proxy *(ImplementationProxy)* to delegate the execution on either the real object *SubjectBody*, or *Channel* if the real object is being coordinated. In the latter case, the *Channel* transfers the execution to the regulator which will then superpose whatever forms of coordination have been required.

Notice that superposing a new connector implies only modifications to the broker, making its proxy become a reference to the object that plays the role of channel. Doing only this minor modification, neither the code of clients nor the code of the broker and of the real object need to be modified in order to accommodate the new behaviour established by adding the connector.

4 Concluding Remarks

In this short paper, we outlined on-going work that explores some of the features made available by coordination models for supporting a discipline of system evolution that is compositional on the evolution of the application domain itself. The approach is based on the explicit identification, as first-class citizens, of the mechanisms that model business rules and other aspects of the application domain that require that the behaviour of its components be coordinated in order to achieve certain effects.

The solution that we found is based on a pattern that captures what in Software Architectures are called *connectors* [1] and implements what in Parallel Program Design is known as *superposition* [8]. In fact, our work consisted in abstracting from the concrete proposals that can be found in the literature, universal principles that capture the essence of these notions as far as their application to our evolutionary approach is concerned, and make it applicable to a wider range of languages and models.

This abstraction process was conducted in two directions. On the one hand, we searched for mathematical patterns that would capture the semantics of the coordination mechanisms that we wanted to make available in OBLOG. On the other hand, because the approach to evolution that we motivated above was developed in response to concrete needs for supporting development work in highly volatile business domains, we had to workout a way of making these coordination mechanisms available in concrete platforms for system development. As a result, a design pattern was developed that can make the proposed coordination mechanisms available over platforms for component-based system development like CORBA, EJB and COM. More specifically, the whole approach is supported by the OBLOG tool and has been tested in a variety of application domains. This design pattern was discussed in section 3. More examples are available in [2], where the notion of *contract* was proposed as an extension to the UML for representing connectors with the semantics discussed herein.

Further work is in progress, both in improving the evolutionary approach and its support through OBLOG, as well as in incorporating other associated mechanisms that we have developed over the proposed mathematical pattern. These include the mechanisms via which reconfiguration can be effectively performed dynamically [9].

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 *UML'99 – Beyond the Standard*, R.France and B.Rumpe (eds), LNCS 1723, Springer Verlag 1999, 566-583.
3. J.L.Fiadeiro and A.Lopes, "Semantics of Architectural Connectors", in *TAPSOFT'97*, LNCS 1214, Springer-Verlag 1997, 505-519.
4. J.L.Fiadeiro and A.Lopes, "Algebraic Semantics of Coordination, or what is in a signature?", in *AMAST'98*, A.Haeberer (ed), Springer-Verlag 1999.
5. J.Goguen, "A Categorical Manifesto", *Mathematical Structures in Computer Science* 1(1), 1991, 49-67.
6. E.Gamma, R.Helm, R.Johnson and J.Vlissides, *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley 1995.
7. D.Gelernter and N.Carriero, "Coordination Languages and their Significance", *Communications ACM* 35, 2, pp. 97-107, 1992.
8. S.Katz, "A Superimposition Control Construct for Distributed Systems", ACM TOPLAS 15(2), 1993, 337-356.
9. M.Wermelinger and J.L.Fiadeiro, "Algebraic Software Architecture Reconfiguration" in *ESEC/FSE'90*, LNCS 1687, Springer-Verlag 1999, 393-409.