Journal of Universal Computer Science, vol. 10, no. 4 (2004), 375-390
submitted: 17/10/03, accepted: 2/2/04, appeared: 28/4/04

Composition Contracts for Service Interaction

Luís Filipe Andrade
(ATX Software S.A., Portugal
`luis.andrade@atxsoftware.com`)

José Luiz Fiadeiro
(University of Leicester, United Kingdom
`jose@fiadeiro.org`)

Abstract: In this paper, we address some of the challenges raised by the emerging service-oriented computing paradigm in what concerns the ability to define dynamic interactions between core services for flexible and agile business processes. We claim that, from this point of view, service interaction and composition is well beyond the reach of object-oriented and component-based techniques. We argue instead for the use of architectural modelling techniques that promote the externalization of coordination mechanisms. We show how what we call composition laws and interfaces can be used to define the coordination logic according to which the behavior of a business process can be described in terms of interactions with given partners. These primitives provide a business modelling level that can be mapped onto the specifications that are being proposed for web services, e.g., BPEL, WS-Coordination or WS-Transaction.

Key Words: architectural connectors, composition, coordination, interaction, service-oriented computing, software architecture, web services
Category: D.2, D.2.10, D.2.11

1 Introduction

In its edition of May 10th 2003, The Economist wrote:

> Computing has certainly got faster, smarter and cheaper, but it has also become much more complex. Ever since the orderly days of the mainframe, which allowed tight control of IT, computer systems have become ever more distributed, more heterogeneous and harder to manage. (. . .) In the late 1990s, the Internet and the emergence of e-commerce "broke IT's back". Integrating incompatible systems, in particular, has become a big headache. A measure of this increasing complexity is the rapid growth in the IT services industry. (. . .)
>
> Computing is becoming a utility and software a service. This will profoundly change the economics of the IT industry. (. . .) For software truly to become a service, something else has to happen: there has to be a wide deployment of web services. (. . .) Applications will no longer be a big chunk of software that runs on a computer but a combination of web services; and the platform for which developers write their programs will no longer be the operating system, but application servers.

The challenges raised by the emerging Service-Oriented Software Development (SOSD) paradigm are, indeed, multiple and the fact that they make news in a journal like The Economist just means that key sectors of the economy and, indeed, the society are feeling threatened. The functioning of the modern society is becoming ever more dependent on the use of software systems. We are by now quite familiar with expressions such as e-Commerce, e-Government, e-Learning, e-Medicine, or e-Science. Unfortunately, this multitude of expressions also reflects a multitude of approaches, each of which tackles the challenges of developing e-services for its own particular area and in its own *ad hoc* way. This fragmentation is real and raises the spectrum of a society whose functioning relies on services that are either not compatible or put together in ways that lead to unpredictable interactions. Is there a justification for the concerns expressed by The Economist? Are there reasons to feel threatened? What are we, scientists, doing about it?

In the literature, web services are being promoted as a technology for dynamic business, the next generation of the Internet culture in which a shift is made from B2C to B2B [IBM, 2003b]. This shift puts an emphasis on program-to-program interaction, which is quite different from the user-to-program interaction that characterized (thin) clients interacting with business applications in the B2C model. In particular, initiatives that were typical of the user side, like searching, have now to be performed on the business side, which means that they need to be supported by some software. Readings on the technologies that have been made available normally emphasize this shift from server-to-server, static, linear interaction to dynamic, mobile and unpredictable interactions between machines operating on a network, and identify it as one of the challenges that needs to be met in full for the architecture to impose itself in its full potential.

The models that have been proposed in the meanwhile, namely the service-oriented architecture based on publishing, finding and binding, address these issues directly in terms of technological solutions that can be supported immediately, at varying levels, by XML, SOAP, WSDL, or UDDI. These efforts are not general enough and lack imagination in exploring the full potential of the service-oriented paradigm, of which web services are only a manifestation. In order to understand what exactly is necessary to support the engineering and deployment of services in general, and, hence, what is still required in terms of research and development effort, we have to characterize exactly what this new architecture is about and how it relates to the software development (SD) methodologies and supporting technologies that are available, namely Component-Based (CB) and Object-Oriented (OO) ones.

The question of determining how exactly object-oriented techniques can contribute to the engineering of services is a fair one. It reflects a (legitimate) concern for the investments that have been made already on object-oriented technologies. It must be made clear to people and, especially, to the companies that want to be part of the "new economy", what kind of investments this new generation of architectures for the web requires before it becomes obsolete. This is particularly important because web services are often presented as a logical evolution of, on the one hand, object-oriented analysis and design and, on the other hand, of components as geared to the deployment of e-business solutions [Yang, 2003].

In this paper, we would like to disagree with this evolutionary view and put forward the case for a new generation of methods and modelling techniques that are interaction-centric and address run-time service composition. We believe that the move from object- to service-oriented systems is a true shift of paradigms; service-oriented computing needs to be fully and formally characterized as a paradigm so that methodologies and supporting technologies can be developed to take maximum profit of its potential. In particular, we shall discuss the role of the notion of contract as an abstraction for the modelling of interconnections. Whereas design-by-contract [Meyer, 1992] supports compile-time object-oriented interactions, i.e., clientship, we propose a notion of composition contract that supports run-time integration of services.

We illustrate the role that architectural primitives based on the separation of coordination from computation can play in tackling some of the challenges raised by SOSD, namely for representing the business processes according to which dynamic interactions are required to be established and evolved. Proposals such as BizTalk [Microsoft, 2000] build on calculi and algebras for concurrency to promote the separation of the definition of business interactions from their implementation. In this respect, we take one step further. It consists of adopting a layer of representation in which interactions become first-class entities and can be modelled in more abstract terms instead of being hard-coded. Finally, we identify the need for further technologies that can complement what coordination and architectures can provide today, namely in what concerns binding and transaction protocols.

The rest of the paper is organized as follows: we first argue on the reasons why we think that service composition is more than object-oriented composition [see Section 2]; we then report on how to use business rules to describe web services compositions [see Section 3]; later, we report on how orchestration can be achieved through binding [see Section 4]; finally, we present our main conclusions [see Section 5].

2 Why Service Composition is Beyond Object-Orientation

The main reason for our disagreement to what is generally perceived as an evolution of object-oriented and component-based software development is precisely the fact that the shift from objects/components to services is reflected fundamentally on the interactions through which services can be composed. Web services have been often characterized as "self-contained, modular applications that can be described, published, located, and invoked over a network, generally the web" [IBM, 2003b]. Building applications is a dynamic process that consists of locating services that provide the basic functionalities that are required, and orchestrating them, i.e. establishing collaborations between them, so that the desired global properties of the application can emerge from their joint behavior.

Integration is another key word in this process, and it is often found in conjunction with orchestration or marshalling: application building in service-oriented architectures is based on the composition of services that have to be discovered and marshalled dynamically. Therefore, one of the features of the service-oriented paradigm is, precisely, the ability that it requires for interconnections to be established and revised dynamically, at run time, without having to suspend execution, i.e., without interruption of service. This is what is usually called late or just-in-time integration, as opposed to compile- or design-time integration.

Therefore, SOSD requires flexible composition mechanisms that are able to make the resulting systems amenable to changes at run time. For this purpose, interactions cannot be hardwired in the code that implements the services, which would lead to systems that are too tightly coupled for the kind of dynamics required by services. Systems need to be in a continuous process of reconfiguration due to the fact that services need to establish, dynamically, the collaborations that allow them to provide the functionalities that are being requested by some other service. If such collaborations are not modelled directly as first-class entities that can be manipulated by a process of dynamic reconfiguration, the overhead that just-in-time integration and other operational aspects of this new architecture represent will not lead to the levels of agility that are required for the paradigm to impose itself.

However, traditionally, interactions in the object-oriented paradigm are based on identities [Kent, 1993], in the sense that, through clientship, objects interact by invoking the methods of specific objects (instances) to get something specific done. This implies that any unanticipated change on the collaborations that an object maintains with other objects needs to be performed at the level of the code that implements that object and, possibly, of the objects with which the new collaborations are established [Shaw, 1996].

On the contrary, interactions in the service-oriented approach should be based on the design of what needs to be done, thus decoupling the what one wants to be done from who does it. This leads directly to the familiar characterisation of web services as late binding or, better, just-in-time binding. It is as if collaborations in the object-oriented paradigm where shifted from instance-to-instance to instance-to-interface, albeit with a more expressive notion of interface. This is why, in our opinion, web services are, indeed, beyond object-oriented methodology and technology, especially in what concerns the support that needs to be provided for establishing and managing interactions.

What is, perhaps, more interesting, is the fact that these limitations can be witnessed in the methodological principles that these paradigms usually subsume. Clientship as in OO/CBSD allows us to decompose the global behavior of a system in a way that mimics typical ways in which people organize themselves in society: by purchasing products directly from the providers. In the context of this societal metaphor the shift from object- to service-oriented interactions mirrors what has been happening already in human society: more and more, business relationships are established in terms of acquisition of services, e.g. 1000 Watts of lighting for `www.welightyourlife.`

`com`, instead of products, e.g., 10 lamps of 100 Watts each from `www.BulbsRUs.com`.

Obviously, object-oriented technology does not prevent at all such flexible modes of interconnections to be implemented. Design mechanisms, making use of event publishing/subscription through brokers and other well-known patterns [Gamma, 1995], have already found their way into commercially available products that support implicit invocation [Notkin et al., 1993] instead of feature calling (explicit invocation). However, solutions based on the use of design patterns are not at the level of abstraction in which the need for integration arises and needs to be managed. Being mechanisms that operate at the design level, there is a wide gap that separates them from the business modelling levels at which orchestration is better perceived and managed.

Indeed, there is a big gap between the high-level specification of interactions and their implementation in any particular technology. This gap is currently being filled by the code of the fine-grain parts that have to glue the domain components with the architectural framework provided by the underlying technology, e.g., CORBA, J2EE, or .NET. Even when using design patterns to structure the code, the final result is a full mix between the code that implements the domain logic and the infra-structural glue code. The lack of a clear separation results in component interactions that are very difficult to maintain and evolve. In summary, design patterns are not first-class citizens as run-time software components, which means that they cannot be taken as structures over which orchestration can be addressed. That is to say, the processes that they represent are not available, a point that has being progressively stressed in the literature [Microsoft, 2000]:

> With the integration and communication infrastructures complete, our applications can now "speak" to other applications over the Internet, but we do not have a good mechanism for telling them when and how to say it. We have no way of representing the process. Today the process is spread throughout the implementation code of every participant involved in the process. This mechanism for representing the process of business interactions is fragile, prone to ambiguities, does not cross organizational boundaries well, and does not scale. The larger the process gets and the more participants that are involved, the more static the process is and the harder it is to propagate changes and new players into the process.

In summary, as a paradigm, SOSD is based on interaction and composition principles that require a fundamental research effort for their full characterisation. Science has to face that a new reality is there as far as software development is concerned, which requires a fresh approach that can meet the challenges that it raises from first principles. OO and CBSD were proposed for controlling the complexity of building big chunks of software, as the article in The Economist says; SOSD will have to control the complexity of managing evolving interactions in large systems.

In this paper, we put an emphasis on the service composition layer of service-oriented architectures, i.e., the level at which business processes can be put together from elementary services. This area has become the realm of recent specifications such as WS-Coordination [IBM, 2003c], WS-Transaction [IBM, 2003d], or the Business Process Execution Language for Web Services (BPEL4WS or BPEL for short) [IBM, 2003a]. The WS jargon is also evolving accordingly. There is now a new terminology that, in some cases, intersects other areas of Computer Science and Software Engineering, which can be a source of confusion. We have tried to avoid these risks by adapting the terminology that we have used in the past to today's WS-audience, but there are cases in which it is difficult to conciliate established practices. This is the case of coordination which, in the literature, became associated in the late 80s with a well-identified area of Computer Science [Gelernter and Carriero, 1992], and is now in the name of a very precise aspect of web services, i.e., WS-Coordination [IBM, 2003c]. The two areas are not unrelated, but WS-Coordination is a very concrete framework that is far from capturing the richness made available through common coordination languages and models.

Therefore, we want to make clear that our purpose is not to provide semantics for specifications such as BPEL, WS-Coordination or WS-Transaction. This is because, on the one hand, such specifications have not stabilized to a point where standards can be meaningfully assigned; they are still very much a moving target, and for very good reasons. On the other hand, and in spite of providing a much needed move beyond the basic framework of SOAP/WSDL/UDDI technologies for publishing, finding, and binding, they are still far from providing the semantic primitives that can address business modelling at a higher-enough level of abstraction. Hence, we would like to persuade the reader to approach the rest of this paper with an open mind and seek inspiration rather than solutions to problems of specific technologies of today.

3 Composing Services According to Business Rules

Services can be seen as granular software components that can be used as building blocks for distributed applications or for the assembly of business processes. They reflect a new architectural approach based on the notion of building applications by discovering and orchestrating network-available services, or just-in-time integration of applications. With SOSD, application design becomes a matter of describing the capabilities of network services to offer certain functionalities through their computational capabilities, and describing the orchestration of these collaborators in terms of mechanisms that coordinate their joint behavior. At run time, application execution is a matter of translating the collaborator requirements into input for a discovery mechanism, locating a collaborator capable of providing the right service, and superposing the coordination mechanisms that will orchestrate the required interactions.

This informal design makes clear that the algebraic methodology that we have been developing around architectural principles in general [Fiadeiro et al., 2003] plays a fundamental role in enabling service-oriented architectures. This methodology is based on the separation between what is concerned with the computations that are responsible for the functionality of the services that they offer and the mechanisms that coordinate the way components interact, a paradigm that has been developed in the context of so-called coordination languages and models [Gelernter and Carriero, 1992].

Our previous work has brought together concepts and techniques from software architectures, e.g., the notion of connector [Allen and Garlan, 1997], parallel program design, e.g., the notion of superposition [Katz, 1993], and distributed systems, e.g., techniques to support the dynamic reconfiguration of a system [Magee and Kramer, 1996]. They are now being integrated as a collection of semantic primitives that support the modelling of interactions that are flexible and more amenable to changes at run time [Andrade and Fiadeiro, 2003a]. These primitives allow for the kind of just-in-time binding required for service-oriented computing.

The underlying methodology of coordination-based development is also essential for service-oriented systems since it externalizes interactions as connectors that can aggregate simpler services dynamically, and encourages developers to identify dependencies between activities in terms of services instead of identities. The identification of the partners to which coordination contracts are applicable is made through interfaces that identify the properties that they need to exhibit rather than the classes to which they have to belong.

Our emphasis in this paper is on the composition aspects that subsume what has been called composition logic in the literature [Yang, 2003], i.e., the way composite services are constructed in terms of constituent services to fulfil specific business goals, or business protocols and processes [Curbera et al., 2003], i.e., the definition of processes or workflows that interact with sets of web services to achieve certain goals in terms of abstract service descriptions, separately from specific deployments.

BPEL puts an emphasis on the definition of service compositions in terms of processes that interact with partners that are external to the composition itself and are identified in terms of abstract interfaces only. This is particularly close to the approach based on coordination contracts and laws that we have been promoting recently for service-oriented business information systems [Andrade and Fiadeiro, 2003b]. Indeed, it is particularly important that we are able to separate the definition of the composition logic, what we would call coordination laws, that capture the business rules according to which complex business activities are put together from more basic services, from the run-time composition of specific services as part of a process that is being executed to fulfil a specific business goal. The purpose of this section is to focus on a coordination model for composing abstract services according to business rules.

To illustrate the kind of approach that we outlined, consider the example of a bank that, on the one hand, wishes to make its financial deals available as services that it publishes on the web and, on the other hand, adopts an internal software development approach based on services that it can rapidly integrate into new applications and, thus, optimize time-to-market.

One of the services that financial institutions have been making available in recent times is what we could call a flexible package, i.e., the ability to coordinate deposits and debits between two accounts, typically a checking and a savings account, so that the balance of one of the accounts is maintained between an agreed minimal and maximal amount by making automatic transfers to and from the other account. The service that is offered is the detection of the situations in which the transfers are required and their execution in transactional mode. Because this behavior should be published as a service that customer applications looking for flexible account management should be able to bind to, the following aspects should be ensured: first, the service cannot be offered for specific accounts. It is the binding process that should be responsible for instantiating the service to the relevant components at execution time. Nor should it be offered for specific object classes: the service itself should be able to be described independently from the technology used by the components to which it will be bound. Instead, it is the find/bind process that should be able to make the adaptation that is needed between the technologies used for implementing the service and the ones used by the components to which it is going to be bound. This adaptation can itself be the subject of an independent publish/find/bind process that takes place at another level of abstraction, or be offered for default combinations by the service itself. This is important to enable the implementation of the service itself to evolve, say in order to take advantage of new technologies, without compromising the bindings that have been made.

Therefore, the description of the partners to which a business process can be bound is made of a coordination interface [Andrade and Fiadeiro, 2003b] in our approach; however, we call it composition interface to avoid confusion with WS-terminology. For instance, in the case of the flexible package, two composition interfaces are required: one catering for the account whose balance is going to be managed, typically a checking account, and the other for the savings account. The trigger/reaction mode of coordination that our approach supports requires that each composition interface identifies which events produced at execution time are required to be detected as triggers for the process to react, and which basic services must be made available for the reaction to superpose the required effects. Notice that this separation is supported in BPEL processes by distinguishing between different kinds of activities that implement interactions between the process and its partners.

The two composition interfaces that we have in mind can be described as follows:

```
composition interface savings-account
import types money;
services
    balance(): money;
    debit(a: money)   post balance(a) = old balance() - a
    credit(a: money)  post balance(a) = old balance() + a
end
```

Notice how the properties of the elementary services that are required are specified in an abstract way in terms of pre- and post-conditions.

```
composition interface checking-account
import types money;
events balance(): money;
services
    balance(): money;
    debit(a:money)   post balance(a) = old balance() - a
    credit(a:money)  post balance(a) = old balance() + a
end
```

The difference between the two interfaces lies in the inclusion of the event in the checking-account. This is because the activities that are required to be observed in order to react by activating an interaction rule are the changes on the balance of the checking account. More specifically, we are requiring from any partner that may be bound to this interface to make changes in the balance available to the flexible-package service.

The second important requirement is that the composite service itself must be described on the basis of these composition interfaces only. This is what, in BPEL, would be called the state and logic necessary for coordinating the interactions between the process and the partners. This composition or coordination logic can be made in terms of what we call a composition law (coordination law in [Andrade and Fiadeiro, 2003b]):

```
composition law flexible-package
interfaces  c: checking-account, s: savings-account
attributes  minimum, maximum: money
interaction rules
    when    c.balance() < minimum
    do      s.debit(min(s.balance(), maximum - c.balance())
            and c.credit(min(s.balance(), maximum - c.balance())
    when    c.balance() > maximum
    do      c.debit(c.balance() - maximum)
            and s.credit(c.balance() - maximum)
end law
```

Besides identifying the composition interfaces, a composition law specifies the rules that define the behavior of the service. Such interaction rules are of the form:

```
when    condition
with    condition
do      set of services
```

Each interaction rule in the `when` clause identifies a trigger to which the process will react, e.g., a change in the balance of the checking-account that takes it out of the bounds. The trigger can be just an event observed directly over one of the partners or a more complex condition built from one or more events. In the `with` clause, we include conditions (guards) that should be observed for the reaction to be performed. If any of the conditions fails, the reaction is not performed and the occurrence of the trigger fails. Failure is handled through whatever mechanisms are provided by the implementation language.

The reaction to be performed by the service is described as a set of elementary activities in the `do` clause. This set may include calls to services provided by one or more of the partners as well as activities that are internal to the coordination logic of the process itself. The whole interaction is handled as a single transaction, i.e., it consists of an atomic event in the sense that the trigger reports a success only if all the activities identified in the reaction execute successfully and the conditions identified in the `with` clause are satisfied.

In what concerns the language in which the reactions are defined, we normally use an abstract notation for defining the synchronisation set as above. This is important for bringing to a more abstract modelling level the definitions of business processes that recent languages for orchestration promote in terms of algebras and models for concurrency, e.g., BizTalk [Microsoft, 2000]. Our opinion and experience is that the architectural modelling level at which we promote the representation of business interactions makes it easier to bridge the gap from the more organisational high-level goals and policies that dictate how business should be run to the choice of particular control and synchronisation structures that can make specific processes run.

When the interfaces are bound to specific partners, their behavior is coordinated by an instance of the law, which establishes at run time what we call a coordination contract [Andrade and Fiadeiro, 1999]. The behavior specified through the rules is superposed on the behavior of the partners without requiring the code that implements them to be changed.

Let us consider another example. Assume that the bank is interested in being able to compute the average balance of given accounts from given dates. This is a service that the bank should be able to bind, at any time, to any given account for monitoring purposes. This service requires the ability to observe the balance of an account, which corresponds to the following composition interface:

```
composition interface account-balance
import types money;
services
    balance(): money
end
```

The corresponding law is:

```
composition law average-balance
interfaces  a: account-balance
attributes  sum: money; days: nat
activities
    report(): money post report = sum / days
    reset() post sum = a.balance() and days = 1
interaction rules
    when    end-of-day
    do      sum = old sum + a.balance(), days = old days + 1
end law
```

We are assuming that `end-of-day` is a global trigger that the service is able to detect in the system through some calendar mechanism that we take for granted. Otherwise, we could define an interface that would need to be instantiated by a specific clock.

Our last example consists of a process that the bank can use to monitor how long the balance of a given account has exceeded a given threshold since it was last reset. For that purpose, and instead of making the calculations at the end of the day, we are going to make the process aware of the debits and credits that are performed in the account:

```
composition interface debits&credits
import types money;
events
    debit(a: money);
    credit(a: money)
services
    balance(): money
end
```

The corresponding law is:

```
composition law counting-excess
interfaces  a: debits&credits
parameter   min: money
attributes  nb-days: nat; last-day: date
activities
    report(): nat post report = nb-days
    reset() post nb-days = 0 and last-day = today()
interaction rules
    when a.credit(n) or a.debit(n)
    do   if a.balance() = min then {
            nb-days = old nb-days + (today() - old last-day)
            and last-day = today()}
end law
```

We are assuming, again, that `today()` is a service that the process is able to use from the system's calendar.

4 Orchestration Through Binding

Laws and interfaces can be used to define the coordination or composition logic according to which the behavior of a business process can be described in terms of interactions with given partners [Yang, 2003]. These define composition types or protocols [Curbera et al., 2003] in the sense that only abstract interfaces are used to identify the partners that can become involved in the process. They do not prescribe when and to which partners services should be bound.

Run-time aspects of interactions need to be subject to rules that aim at enforcing given policies of organisations over the way they wish or are required to see their businesses conducted, e.g., through legislation. For this purpose, we provide a modelling primitive called composition processes by means of which reconfiguration capabilities can be automated for dynamic integration, both in terms of *ad hoc* and programmed bindings through which the run-time orchestration of services can be specified as a means of achieving specific operational business goals. For instance, the bank may wish to offer VIP clients the possibility of having interests awarded on the average balance for the number of days the account has been above an agreed threshold. This can be achieved by composing the services that we described in the previous paragraph.

```
process interest-on-average(a: account, c: customer)
attributes interest: rate; performance: nat
activities
subscribe (r: rate, m: money):
    pre: c.owns(a) & isVIP(c)
    do:  bind av: average-balance->a,
         bind ct: counting-excess->a,
         av.reset(),
         ct.reset(),
         ct.min = m,
         interest = r,
         performance = 0
earn:
    do: a.credit(r * ct.report() / 365 * av.report()),
        av.reset(),
        ct.reset(),
        performance = old performance + ct.report()
re-rate(r: rate):
    pre: performance > 100
    do:  earn,
         interest = r,
         performance = 0
re-min(m: money):
    do: earn,
        ct.min = m
rules
    yearly: when end-of-year do earn
end process
```

Such a process defines a set of activities through which services can be bound to given partners in the system and their activities orchestrated to define required business activities such as renegotiating rates (`re-rate`) or crediting earned interest (`earn`). Rules can also be defined that capture autonomous behavior such as automatically crediting earned interest at the end of the year (`yearly`).

It is important to draw attention to the bind instructions. The emphasis that we put on the role of composition interfaces in the previous section should have made clear how important the binding process is for our approach. Indeed, our answer to the challenges raised by SOSD is based on an architecture and development process that promotes the separation of the key factors that drive integration and the need for orchestration [see Figure 1]:

1. Separation of the coordination mechanisms that regulate the way basic services in-

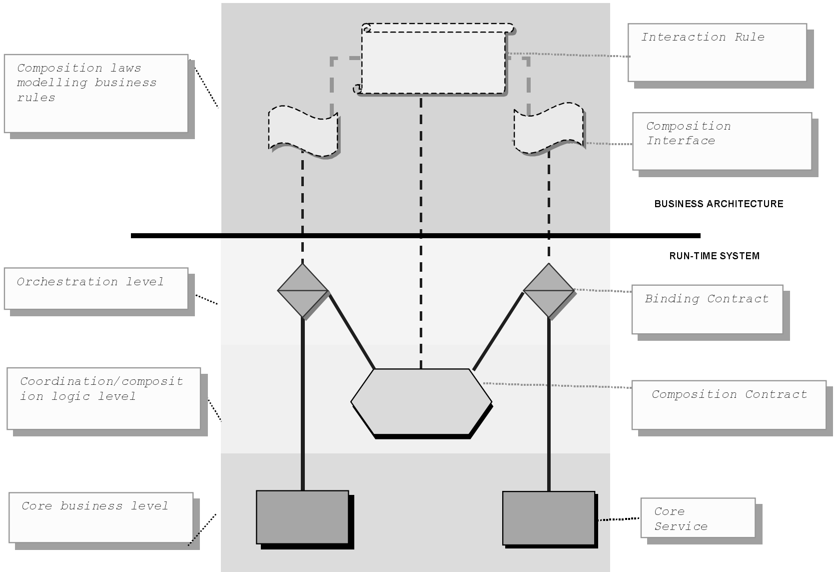

Figure 1: Separating business process configuration, coordination/composition logic, and core service computation

teract to enforce current business rules and activities, from the computations that realize the functionalities that are offered by core business services, i.e., coordination/composition technologies.

2. Separation of the business architecture according to which an application is structured from the particular partners and interconnections that constitute the run-time configuration over which it is implemented, i.e., binding technologies.

In both cases, the separation leads to the definition of explicit primitives whose instances can be superposed dynamically to support just-in-time integration, i.e., composition contracts and binding contracts, respectively. Composition contracts are instances of composition laws which represent the business rules that are embodied in a service; composition contracts superpose interconnections among a set of partners regulated according to a set of trigger/reaction rules specified in the law from which the contract derives.

The level of insulation between the business model (where the law resides) and the implementation level (where the contract resides) is achieved by (i) defining the law independently from the partners to which it can be applied by identifying the features and properties that partners need to exhibit to become coordinated according to that law; (ii) using binding contracts to instantiate composition interfaces and rules of given laws with partners discovered in given locations of the net and interconnections that reflect the properties of the distribution network. Hence, the use of binding contracts makes it possible for partners and contracts to be replaced in order to reflect mobility without interfering with the business architecture. This ensures agility for operating in web-based environments (just-in-time integration). These binding technologies are not yet developed at the same level as the coordination technologies. As far as our work is concerned, they are still in their infancy.

One should mention, however, related programming paradigms that promote the separation of concerns and automated generation of code, e.g., generative programming. For instance, Aspect-Oriented Programming (AOP) [Elrad et al., 2001] encompasses a wide range of programming techniques to enable the separation of cross-cutting concerns that do not conveniently fit into the dominant hierarchy, e.g., objects or components. Languages such as Xerox PARC's AspectJ and IBM's Hyper/J are now widely used and have been extensively tested on large-scale complex systems with very positive results. The code they produce is structured cleanly, has minimal duplication, and can be maintained easily. Given the nature of web services, it is inevitable that such techniques will have a significant part to play in SOSD [Corchuelo et al., 2003]. The customized services must be able to adapt to, and interwork with other services or service components. The elegant structuring mechanisms afforded by techniques like AOP can be used to facilitate this adaptation and interworking. It should be noted that the main focus of these studies has been AspectJ, due to its high level of maturity and excellent tool support. However, the nature of AspectJ is, at present, largely static, in notable contrast to the highly dynamic nature of web services.

The Model Driven Architecture (MDA) initiative [Kleppe et al., 2003] proposed by the OMG is based on the separation of the specification of system functionality from the specification of the implementation of that functionality on a specific platform. It aims at making the software assets more resilient to changes caused by the emerging technologies and makes the role of modelling and models in the current software development much more important. However, it still has to prove that the model supports the just-in-time generation of code required by SOSD.

5 Concluding Remarks

We are well aware that this paper started with strong claims about the need for Science to respond to the challenge presented by SOSD as a new paradigm and, so far, not a single formula was shown. This is because the main aims of the paper are, on the one hand, to demonstrate that SOSD cannot be dismissed as just an evolution of OO/CBSD since it requires a completely different approach to interactions, and, on the other hand, that some of the challenges that it raises can be met using architectural modelling principles and techniques based on coordination technologies.

As far as foundations and semantics are concerned, the concepts and techniques that we presented are firmly grounded on mathematics. Indeed, we have already shown elsewhere how the key separation between composition interfaces and laws, and the mechanisms through which bindings can be formalized, can be captured in an algebraic framework consisting of [Fiadeiro et al., 2003]:

1. A category DES of designs in which systems of interconnected components are modelled through diagrams and colimits capture emergent behavior.

2. A functor $sig: DES \to SIG$ that maps (service) designs onto (composition) interfaces, forgetting their computational aspects, satisfying the following key properties: sig is faithful, lifts colimits of well-formed configurations, and has discrete structures.

3. A category r-DES with the same objects as DES (designs) but whose morphisms support binding through refinement relations.

This formalisation relates directly to the semantics of so-called architectural connectors [Allen and Garlan, 1997] and the notion of superposition [Katz, 1993] since they have been used to express incremental evolution of systems. There is no space left in this paper for showing how this semantics applies directly to SOSD but, by now, the reader will have acquired the motivation that can guide him or her through our publications on the subject, which can be found at `http://www.fiadeiro.org/jose/CommUnity`.

This semantics has also been used for showing how the proposed separation of concerns and the dynamic superposition of composition contracts can be implemented in Java-based platforms [Andrade et al., 2002]. One of the aspects that still require further work are the binding technologies and their use in composition contexts. This is an area of current research. As far as our work is concerned, work on publishing technologies is lagging even further behind. Finally, we should also mention the opportunities that we see in exploring other transaction protocols [Little, 2003] in the definition of our interaction rules.

Acknowledgements

This work was partially supported by the Marie-Curie TOK-IAP scheme under contract 3169 $\mathcal{Leg}\mathbf{2}^{\text{net}}$.

References

[Allen and Garlan, 1997] Allen, R. and Garlan, D. (1997). A formal basis for architectural connectors. *ACM TOSEM*, 6(3):213–249.

[Andrade and Fiadeiro, 1999] Andrade, L. and Fiadeiro, J. (1999). Interconnecting objects via contracts. In France, R. and Rumpe, B., editors, *Proc. of UML'99 - Beyond the Standard*, number 1723 in LNCS, pages 566–583. Springer.

[Andrade and Fiadeiro, 2003a] Andrade, L. and Fiadeiro, J. (2003a). Architecture based evolution of software systems. In Bernardo, M. and Inverardi, P., editors, *Proc. of Formal Methods for Software Architectures*, number 2804 in LNCS, pages 148–181. Springer.

[Andrade and Fiadeiro, 2003b] Andrade, L. and Fiadeiro, J. (2003b). Service-oriented business and system specification: Beyond object-orientation. In Kilov, H. and Baclwaski, K., editors, *Practical Foundations of Business and System Specifications*, pages 1–23. Kluwer Academic Publishers.

[Andrade et al., 2002] Andrade, L., Fiadeiro, J., Gouveia, J., Koutsoukos, G., and Wermelinger, M. (2002). Coordination for orchestration. In Arbab, F. and Talcott, C., editors, *Proc. of COORDINATION'02*, number 2315 in LNCS, pages 5–13. Springer.

[Corchuelo et al., 2003] Corchuelo, R., Pérez, J., and Ruiz-Cortés, A. (2003). Aspect-oriented interaction in multi-organizational web-based systems. *Computer Networks*, 41(4):385–406.

[Curbera et al., 2003] Curbera, F., Khalaf, R., Mukhi, N., Tai, S., and Weerewarana, S. (2003). The next step in web services. *Communications of the ACM*, 46(10):41–47.

[Elrad et al., 2001] Elrad, T., Filman, R., and Bader, A. (2001). Special issue on aspect oriented programming. *Communications of the ACM*, 44(10).

[Fiadeiro et al., 2003] Fiadeiro, J., Lopes, A., and Wermelinger, M. (2003). A mathematical semantics for architectural connectors. In Backhouse, R. and Gibbons, J., editors, *Generic Programming*, number 2793 in LNCS, pages 190–234. Springer.

[Gamma, 1995] Gamma, E. (1995). *Design patterns: elements of reusable object–oriented software*. Addison-Wesley professional computing series. Addison-Wesley.

[Gelernter and Carriero, 1992] Gelernter, D. and Carriero, N. (1992). Coordination languages and their significance. *Communications ACM*, 35(2):97–107.

[IBM, 2003a] IBM (2003a). Business process execution language for web services, version 1.1. `http://www.ibm.com/developerworks/web/library/ws-bpel/`.

[IBM, 2003b] IBM (2003b). Web services architecture overview: the next stage of evolution for e-business. `http://www.ibm.com/developerworks/web/library/w-ovr/`.

[IBM, 2003c] IBM (2003c). Web services coordination, version 1.0. `http://www.ibm.com/developerworks/web/library/ws-coor/`.

[IBM, 2003d] IBM (2003d). Web services transaction, version 1.0. `http://www.ibm.com/developerworks/web/library/ws-transpec/`.

[Katz, 1993] Katz, S. (1993). A superimposition control construct for distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(2):337–356.

[Kent, 1993] Kent, W. (1993). Participants and performers: A basis for classifying object models. In *Proc. of OOPSLA 1993 Workshop on Specification of Behavioral Semantics in Object-Oriented Information Modeling*.

[Kleppe et al., 2003] Kleppe, Z., Warmer, J., and Bast, W. (2003). *MDA Explained: The Model Driven Architecture —Practice and Promise*. Addison-Wesley.

[Little, 2003] Little, M. (2003). Transactions and web services. *Communications of the ACM*, 46(10):49–54.

[Magee and Kramer, 1996] Magee, J. and Kramer, J. (1996). Dynamic structure in software architectures. In *Proc. of the 4th Symposium on Foundations of Software Engineering*, pages 3–14. ACM Press.

[Meyer, 1992] Meyer, B. (1992). Applying design by contract. *IEEE Computer*, 25(10):40–51.

[Microsoft, 2000] Microsoft (2000). *BizTalk Orchestration —a new technology for orchestrating business interactions*. Microsoft Press.

[Notkin et al., 1993] Notkin, D., Garlan, D., Griswold, W., and Sullivan, K. (1993). Adding implicit invocation to languages: Three approaches. In Nishio, S. and Yonezawa, A., editors, *Object Technologies for Advanced Software*, number 742 in LNCS, pages 489–510. Springer.

[Shaw, 1996] Shaw, M. (1996). Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status. In Lamb, D., editor, *Studies of Software Design*, number 1078 in LNCS. Springer.

[Yang, 2003] Yang, J. (2003). Web service componentization. *Communications of the ACM*, 46(10):35–40.