Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 212 (2008) 207–223
www.elsevier.com/locate/entcs

Services and Contracts: Coalgebraically[1]

Sun Meng[2]

CWI, Kruislaan 413, Amsterdam, The Netherlands

Abstract

The popularity of service-oriented computing has not been accompanied by the necessary formalization of the notions being involved. This paper focuses on the development of a coalgebraic framework to support service-oriented application design. In this paper, the concepts are separated into three hierarchies – interfaces, contracts and services. Interfaces are specified by functors, and services are shown to be coalgebras of such functors, which should satisfy the axioms given in corresponding contracts. Different interfaces, contracts and services are related respectively by the morphisms between them. And the notion of bisimulation for services is derived from service morphisms, which captures the observational equivalence of services.

Keywords: interface, contract, service, coalgebra, composition

1 Introduction

Service-oriented Computing (SOC) [11,16] has now become the prominent paradigm for distributed computing and e-commerce, creating opportunities for service providers and application developers to use services as fundamental elements in their application development processes. It provides a mean to design Service-oriented Applications (SOAs) that span organizations and computing platforms by exploiting and composing services available over the network. Nowadays, an increasing number of companies and organizations implement only their core businesses and use other application services over the Internet to support their needs. Services are platform- and network-independent components that support rapid low-cost composition of distributed applications and can be described, published, discovered, and loosely coupled in novel ways.

The notion of service has been widely used in application development, especially due to the use of service technologies such as the Web Services Description Language (WSDL) [21], Universal Description, Discovery, and Integration (UDDI) [19], Simple

[1] The work reported in this paper is supported by a grant from the GLANCE funding program of the Dutch National Organization for Scientific Research (NWO), through project CooPer (600.643.000.05N12).

[2] Email: M.Sun@cwi.nl

doi:10.1016/j.entcs.2008.04.063

Object Access Protocol (SOAP) [20], which intend to provide languages that allow easy integration of services. Some other initiatives such as the Business Process Execution Language for Web Services (BPEL4WS or BPEL) [2], are focused on representing service compositions where flow of a process and bindings between services are known a priori. Despite all these efforts, composition of services still remains a highly complex task, and automatic composition of services is a critical problem. Conspicuously, the principles of component-based software development (CBSD) [18] are not yet integrated in the various methods that currently exist for custom composition of services.

Until now, most of the existing methods only describe how to communicate with a service (the syntax), but not the expected effects of such communications (the semantics). Therefore, the information that can be obtained from the service descriptions given in these approaches is limited to the signature of operations provided by the services. In particular, no information about the effect of invoking the operations can be obtained from the service descriptions.

It has been widely recognized that SOAs development is a rather intricate activity and there is still no general agreement about the formal foundation of SOC. Thus the need for a formal foundation of services was identified by many researchers. Especially, the development of a general theoretical framework for SOA development is one of the few prominent challenges in computer science. In this paper, we build a formal framework to support SOA design which separates the three concepts – interfaces, contracts and services. The semantics of services and contracts are given by coalgebras, which are the formal duals of algebras [10,17]. In coalgebras the state spaces are taken as black boxes to which one can only have limited access via specified operations. This aspect is one main characterization of components (and also services) and thus makes coalgebras appropriate for specifying semantics of services. The notion of bisimulation models the observational indistinguishability of service behavior for two services with the same interfaces.

Note that the notion of service is distinguished from that of contract in this paper. The latter is the specification of services, which specifies both the interface of the service and the effects of the service behavior. A service is a model of the contract which meets all the requirements in the contract. Following the unification principle of system modularization and composition in category theory proposed by J. Goguen [8], which has been used by J. Fiadeiro and T. Maibaum in parallel program design [7], we build three categories – the categories of interfaces, contracts and services respectively, and show the relation between them. In the proposed categorical framework, we show how services can be composed together.

The organization of this paper is as follows: Section 2 introduces the concepts of interfaces, contracts and services. Section 3 and Section 4 show how we can put contracts and services together to form more complex contracts and services. Relationships between services, including bisimulation and bisimulation up to a natural transformation, are presented in Section 5. Section 6 investigates the refinement relations for contracts and services respectively. Section 7 presents a family of operators on composing services. Final remarks appear in Section 8.

2 Interfaces, Contracts and Services

In a service-oriented application, interfaces are used to model the seams between different services. A service encapsulates a number of operations through a public interface which provides limited access to the service. By declaring an interface, one can specify the desired behavior type of a service independent of its implementation.

2.1 Interfaces

Taking into account the nature of services, an interface should be comprised of three kinds of *features*: the *type* feature, the *variable* feature and the *value* feature. The type feature includes the information that is state-independent and gives the data context in which the service is placed. The variable feature denotes a family of variables and keeps the information that is state dependent. The value feature accounts for the observations and actions that the service may perform.

Definition 2.1 An interface is a triple $I = (\mathbf{T}, \mathbf{V}, \mathbf{M})$ where

- $\mathbf{T}$ denotes a family of types;
- $\mathbf{V}$ denotes a family of variables;
- $\mathbf{M}$ denotes a family of observations and actions, each observation or action may have parameters.

In this definition, $\mathbf{T}$ is a family of sets being used in the other two counterparts, and the elements of these sets are designated by types. For any interface I there is always a type $T_C \in \mathbf{T}$ which denotes the state space of the corresponding service. $\mathbf{V}$ is a set of variable definitions. Every variable is an entity in which values of a particular type can be stored and every variable definition in $\mathbf{V}$ has the form $x : T$ where x and $T \in \mathbf{T}$ represent the name and type of the variable respectively. Every variable defined in $\mathbf{V}$ of an interface must have distinct name. All the variables in $\mathbf{V}$ are accessible by clients of the interface. $\mathbf{M}$ is a set of method definitions. Every method in $\mathbf{M}$ denotes a possible operation provided by the interface and a method declaration in $\mathbf{M}$ can be an *action*, which has the form $m : T_C \times S \to T_C \times O$, or an *observer* with the form $o : T_C \times S' \to O'$. The types S, S' and O, O' denotes the input and output parameter types of the corresponding methods respectively [3]. The method name m and the parameter types together gives the signature of the method. In an interface, two methods with the same name can be declared, but they must have different signature.

Often a service is specified as a collection of interfaces over a shared state space, each of which exhibiting some operations. Merging the interfaces together results in a service which provides the different kinds of operations simultaneously by the resulted interface.

[3] In this paper we will focus on the case that the type S' is instantiated with the singleton $\mathbf{1}$, which collapse part of the observation structure. Since $T_C \times \mathbf{1} \cong T_C$, an observer can be equally represented by a function with the form $o : T_C \to O'$, which is usually called an *attribute* in the object-oriented programming paradigm.

Definition 2.2 For two interfaces $I_1 = (\mathbf{T}_1, \mathbf{V}_1, \mathbf{M}_1)$ and $I_2 = (\mathbf{T}_2, \mathbf{V}_2, \mathbf{M}_2)$, if no variable is declared simultaneously with different types in both $\mathbf{V}_1$ and $\mathbf{V}_2$, then I_1 and I_2 can be merged together and their merge $I = I_1 \uplus I_2$ is defined as $I = (\mathbf{T}, \mathbf{V}, \mathbf{M})$, where

- $\mathbf{T} = \mathbf{T}_1 \cup \mathbf{T}_2$;
- $\mathbf{V} = \mathbf{V}_1 \cup \mathbf{V}_2$;
- $\mathbf{M} = \mathbf{M}_1 \cup \mathbf{M}_2$.

Theorem 2.3 *The merge of interfaces satisfies:*

- $I \uplus I = I$.
- $I_1 \uplus I_2 = I_2 \uplus I_1$.
- $I_1 \uplus (I_2 \uplus I_3) = (I_1 \uplus I_2) \uplus I_3$.

Interface can be built in successive steps, at each step adding declarations with the **extend** operator.

Definition 2.4 For a given interface $I_1 = (\mathbf{T}_1, \mathbf{V}_1, \mathbf{M}_1)$, if no variable in $\mathbf{V}_1$ is redeclared in $\mathbf{V}_2$, then the expression

$$\textbf{extend } I_1 \textbf{ with } (\mathbf{T}_2, \mathbf{V}_2, \mathbf{M}_2)$$

defines a new interface $I = (\mathbf{T}, \mathbf{V}, \mathbf{M})$ where

- $\mathbf{T} = \mathbf{T}_1 \cup \mathbf{T}_2$;
- $\mathbf{V} = \mathbf{V}_1 \cup \mathbf{V}_2$;
- $\mathbf{M} = \mathbf{M}_1 \cup \mathbf{M}_2$.

Theorem 2.5 *The extension of interfaces satisfies:*

- **extend** I_1 **with** $I_2 = I_1 \uplus I_2$.
- **extend** I_1 **with** $(I_2 \uplus I_3) =$ **extend** $(I_1 \uplus I_2)$ **with** I_3.
- **extend** I_1 **with** $(I_2 \uplus I_3) = (\textbf{extend } I_1 \textbf{ with } I_2) \uplus I_3$.

Interfaces can also be built by hiding some features from existing interfaces.

Definition 2.6 For a given interface $I_1 = (\mathbf{T}_1, \mathbf{V}_1, \mathbf{M}_1)$, if $\mathbf{T}_2 \subseteq \mathbf{T}_1$, $\mathbf{V}_2 \subseteq \mathbf{V}_1$, $\mathbf{M}_2 \subseteq \mathbf{M}_1$, and no types in $\mathbf{T}_2$ is used in $\mathbf{V}_1 \setminus \mathbf{V}_2$ and $\mathbf{M}_1 \setminus \mathbf{M}_2$, then

$$\textbf{hide } (\mathbf{T}_2, \mathbf{V}_2, \mathbf{M}_2) \textbf{ in } I_1$$

defines a new interface $I = (\mathbf{T}, \mathbf{V}, \mathbf{M})$ where

- $\mathbf{T} = \mathbf{T}_1 \setminus \mathbf{T}_2$;
- $\mathbf{V} = \mathbf{V}_1 \setminus \mathbf{V}_2$;
- $\mathbf{M} = \mathbf{M}_1 \setminus \mathbf{M}_2$.

2.2 Contracts and Services

The main aspect of a service that we wish to capture coalgebraically is that it has a hidden state space, which is only accessible via limited operations. Therefore, services will be naturally presented as coalgebraic models of *contracts*, which are the interface specifications.

We first come to the notion of contracts. It is clear that for most services it is necessary to have developers working on different services simultaneously to reduce overall schedule time. To make developers work with a reasonable degree of independence, the services and interfaces should be identified and specified with no ambiguity. A contract is such a clear, unambiguous statement, which says precisely what the essential properties of the services (interfaces) are.

Definition 2.7 A contract is a triple $Ctr = (I, \mathbf{L}, \mathbf{A})$ where

- $I = (\mathbf{T}, \mathbf{V}, \mathbf{M})$ is an interface;
- $\mathbf{L} = \mathbf{local}\ (\mathbf{T}_L, \mathbf{V}_L, \mathbf{M}_L)$ denotes a collection of local declarations;
- $\mathbf{A} = (\mathbf{X}, \mathbf{C})$ where $\mathbf{X}$ denotes a collection of axioms specifying properties of behavior of the services implementing the contract, and $\mathbf{C}$ denotes a collection of creating conditions for specifying the properties hold for initially created services, denoted by init.

For an arbitrary contract Ctr as defined in Definition 2.7, we can always associate a functor F with its interface I. Suppose there are n actions $m_1, m_2, \cdots, m_n \in \mathbf{M}$, with the form $m_i : T_C \times S_i \to T_C \times O_i$ for $i = 1, 2, \cdots, n$ respectively, and k observers $o_1, o_2, \cdots, o_k \in \mathbf{M}$, with the form $o_j : T_C \to O'_j$ for $j = 1, 2, \cdots, k$ respectively. Then let $A = \prod_{1 \le j \le k} O'_j$ be the cartesian product of the result types of the k different, but simultaneously available, observers $o_1, o_2, \cdots, o_k$, $S = \sum_{1 \le i \le n} S_i$ and $O = \sum_{1 \le i \le n} O_i$ be the sum of the input and output parameters of the n actions $m_1, m_2, \cdots, m_n$ respectively, we can get the functor

$$\mathsf{F} \;=\; A \times (- \times O)^S \tag{1}$$

The shape of this functor expresses the way the state of the corresponding service is accessed through observers and, on the other hand, how it evolves, through actions.

The behavior of a service specified by the contract with interface F given in (1) is totally deterministic. However, there also may be other possibilities, capturing more complex behavior features. For example, one may know how to get output from input but not in all cases. Then the functor is replaced by

$$\mathsf{F} \;=\; A \times (- \times O + \mathbf{1})^S$$

which results a service whose behavior is *partial*. For a given input, it returns either a valid result or an exception value. One may also be uncertain of the result of service behavior, in the sense that the evolution of the service may be nondeterministic. In

this case the functor is typed as

$$\mathsf{F} \;=\; A \times \mathcal{P}(- \times O)^S$$

where $\mathcal{P}$ denotes the finite powerset monad. This means that the computation of the service action will not simply produce an output and a successor state, but a $\mathcal{P}$-structure of such pairs.

Keeping in mind the discussion above, we may parameterize the interface functor F by a strong monad B, acting as its behavior model and abstracting away from any concrete behavior model.

$$\mathsf{F} \;=\; A \times \mathsf{B}(- \times O)^S \tag{2}$$

Note that here we use the exponential Y^X representing the set of functions from X to Y. For an arbitrary action $m : T_C \times S \to \mathsf{B}(T_C \times O)$, which produces a B-structure as the result of its computation, we can get its transpose $\overline{m} : T_C \to \mathsf{B}(T_C \times O)^S$, whose signature is specified by the functor F as given in (2).

The logic being used in axioms and creating conditions is equational logic. For a type T, the terms over T are built from constants, λ-abstraction, application, tuples, case distinction and (co)projections. Atomic formula are equalities $t_1 = t_2$ for terms t_1 and t_2. Formulas are closed under the logic connectives and quantifiers. If there is a variable $\mathsf{init} : T_C$ in a formula, then the formula is a creating condition. Otherwise, it is an axiom.

Definition 2.8 Consider a contract as defined in Definition 2.7, with functor F associated with the interface. A service satisfying the contract consists of the following elements:

- A carrier set U, giving an interpretation of the state space;
- A transition structure $\alpha : U \to \mathsf{F}(U)$ interpreting the methods of the contract in $\mathbf{M}$ in such a way that the axioms in $\mathbf{X}$ are satisfied;
- An initial state $u_0 \in U$ which satisfies the conditions in $\mathbf{C}$.

Therefore, a service satisfying a contract is given by a seeded F-coalgebra of the "public" interface. Furthermore, it should behave like a F'-coalgebra where F' is associated with all the methods in both $\mathbf{M}$ and $\mathbf{M}_L$: there is a seeded F'-coalgebra $(V, \beta : V \to \mathsf{F}'(V), v_0)$ such that u_0 and v_0 can not be distinguished from each other by the observers and actions in $\mathbf{M}$. If a service p satisfies a contract Ctr, we say that p is an *implementation* of Ctr.

In this picture, a service implements the operations specified in the contracts. And a service can be instantiated to different instances, which may contain the particular states that can be inspected via the attributes and methods implemented by the service.

3 Interface and Contract Morphisms

In the previous section we discussed interfaces, contracts as interface specifications, and single services as instantiations of contracts. One of the most important characterizations of SOC technology is the support for building complex applications by composition of services. Formal tools for this purpose can be borrowed from category theory, the idea is to put service units as coalgebras together to make applications, which are still coalgebras. Hence, we need to provide a notion of morphism between coalgebras of different functors (i.e. services with different interfaces) and build a category of these coalgebras. Before that, we first have a look at the interface morphisms and contract morphisms.

Definition 3.1 Given two interfaces $I_1 = (\mathbf{T}_1, \mathbf{V}_1, \mathbf{M}_1)$ and $I_2 = (\mathbf{T}_2, \mathbf{V}_2, \mathbf{M}_2)$, a morphism $\phi : I_1 \to I_2$ consists of

- A morphism of type $\phi_{\mathbf{T}} : \mathbf{T}_1 \to \mathbf{T}_2$ such that $\forall T \in \mathbf{T}_1$, $\phi_{\mathbf{T}}(T) \in \mathbf{T}_2$, and $\phi_{\mathbf{T}}$ is distributive over product and sum of types: $\phi_{\mathbf{T}}(\prod_{1\leq i\leq n} T_i) = \prod_{1\leq i\leq n} \phi_{\mathbf{T}}(T_i)$ and $\phi_{\mathbf{T}}(\sum_{1\leq i\leq n} T_i) = \sum_{1\leq i\leq n} \phi_{\mathbf{T}}(T_i)$;
- $\forall v : T$ in $\mathbf{V}_1$, a variable $\phi_{\mathbf{V}}(v) : \phi_{\mathbf{T}}(T)$in $\mathbf{V}_2$;
- $\forall m \in \mathbf{M}_1$ with the signature $m : T \to T'$, a method $\phi_{\mathbf{M}}(m) : \phi_{\mathbf{T}}(T) \to \phi_{\mathbf{T}}(T')$ in $\mathbf{M}_2$.

Proposition 3.2 *Interfaces and interface morphisms together constitute a category* **Intf**. *This category is finitely cocomplete and has* $(\emptyset, \emptyset, \emptyset)$ *as the initial interface.*

Once the interface morphism $\phi : I_1 \to I_2$ is given, we can always derive a morphism between the functors F_1 and F_2 corresponding to the two interfaces, i.e., a natural transformation $\varphi : \mathsf{F}_1 \to F_2$.

Given an interface morphism $\phi : I_1 \to I_2$ and formula $t_1 = t_2$, the formula transformation $\phi(t_1 = t_2)$ associated with ϕ is $\phi(t_1) = \phi(t_2)$. And formula transformation is closed under logic connectives and quantifiers.

Definition 3.3 Given two contracts $Ctr_1 = (I_1, \mathbf{L}_1, \mathbf{A}_1)$ and $Ctr_2 = (I_2, \mathbf{L}_2, \mathbf{A}_2)$, a contract morphism $\psi : Ctr_1 \to Ctr_2$ is an interface morphism $\phi : I_1 \to I_2$ that preserving the properties specified by $\mathbf{A}_1$, i.e. for every formula f in $\mathbf{A}_1$, we can get the formula transformation $\phi(f)$ from $\mathbf{A}_2$.

Proposition 3.4 *Contracts and contract morphisms together constitute a category* **Ctr**. *This category is finitely cocomplete.*

Contracts may be built incrementally. The idea is that we build a new contract by putting together two smaller ones, which may share some features.

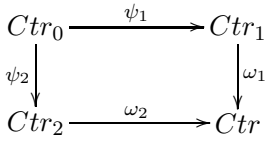
Definition 3.5 The merge of two contracts $Ctr_1 = (I_1, \mathbf{L}_1, \mathbf{A}_1)$ and $Ctr_2 = (I_2, \mathbf{L}_2, \mathbf{A}_2)$ is denoted by

$$Ctr = Ctr_1 \parallel_{Ctr_0} Ctr_2$$

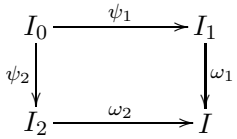
where $Ctr_0 = (I_0, \mathbf{L}_0, \mathbf{A}_0)$ is a subcontract of both Ctr_1 and Ctr_2, denoting their shared parts. Generally, $Ctr = (I, \mathbf{L}, \mathbf{A})$ is defined by

- $I = I_1 \uplus I_2$;
- $\mathbf{L} = \mathbf{L}_1 \cup \mathbf{L}_2$;
- $\mathbf{A} = \mathbf{A}_1 \cup \mathbf{A}_2$.

Ctr is in fact the pushout of the following diagram, where ψ_1 and ψ_2 denote the inclusion as contract morphisms from Ctr_0 to Ctr_1 and Ctr_2 respectively.



At the interface level, if $Ctr = (I, \mathbf{L}, \mathbf{A})$ is the result of merging two contract $Ctr_1 = (I_1, \mathbf{L}_1, \mathbf{A}_1)$ and $Ctr_2 = (I_2, \mathbf{L}_2, \mathbf{A}_2)$, while $Ctr_0 = (I_0, \mathbf{L}_0, \mathbf{A}_0)$ is a subcontract of both Ctr_1 and Ctr_2, denoting their shared parts, then I can be defined in terms of a pushout on the category **Intf** as shown in the following diagram:



If Ctr_0 is an empty contract, i.e., there is no common part of Ctr_1 and Ctr_2, then we can simplify the notation and denote the merge of Ctr_1 and Ctr_2 as

$$Ctr = Ctr_1 \parallel Ctr_2$$

Theorem 3.6 *The merge of contracts satisfies:*

- $Ctr \parallel_{Ctr} Ctr = Ctr$.
- $Ctr_1 \parallel_{Ctr_0} Ctr_2 = Ctr_2 \parallel_{Ctr_0} Ctr_1$.
- $(Ctr_1 \parallel_{Ctr_0} Ctr_2) \parallel_{Ctr_0} Ctr_3 = Ctr_1 \parallel_{Ctr_0} (Ctr_2 \parallel_{Ctr_0} Ctr_3)$.

And it is easy to get the following corollary:

Corollary 3.7 *The merge of contracts satisfies:*

- $Ctr \parallel Ctr = Ctr$.
- $Ctr_1 \parallel Ctr_2 = Ctr_2 \parallel Ctr_1$.
- $(Ctr_1 \parallel Ctr_2) \parallel Ctr_3 = Ctr_1 \parallel (Ctr_2 \parallel Ctr_3)$.

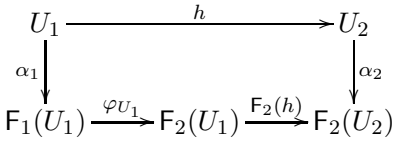
4 Service Morphisms

The morphism between services can be defined as follows:

Definition 4.1 Given two contracts Ctr_1 and Ctr_2, with functors F_1 and F_2 associated with their interfaces respectively. For services $p_1 = (U_1, \alpha_1, u_0^1)$ satisfying

Ctr_1 and $p_2 = (U_2, \alpha_2, u_0^2)$ satisfying Ctr_2. A service morphism ρ from p_1 to p_2 consists of the following elements:

- a natural transformation $\varphi : \mathsf{F}_1 \to \mathsf{F}_2$ derived from the corresponding interface morphism.
- a morphism $h : U_1 \to U_2$ which preserves the initial state $h(u_0^1) = u_0^2$ and makes the following diagram commute:



Proposition 4.2 *Services and service morphisms together constitute a category* **Cop**. *The signature functor* G : **Cop** $\to$ **Intf** *which maps every service (seeded* F*-coalgebra) to its interface (the functor* F*) and every arrow* (φ, h) *to* φ *is a cofibration.*

Proof. The composition of two arrows $\rho_1 = (\varphi_1, h_1) : p_1 \to p_2$ and $\rho_2 = (\varphi_2, h_2) : p_2 \to p_3$ is $\rho_2 \circ \rho_1 = (\varphi_2 \circ \varphi_1, h_2 \circ h_1) : p_1 \to p_3$. Associativity of composition is inherited from that in **Set** and **Cat**. It is even easier to show that there is an identity morphism for any service $p = (U, \alpha : U \to \mathsf{F}(U), u_0)$ which is defined by (id_{F}, id_U) where id_{F} is the identity natural transformation from F to itself and id_U is the identity function on the state space U. Therefore, **Cop** forms a category. For every arrow $\varphi : \mathsf{F}_1 \to \mathsf{F}_2$, and the service $p = (U, \alpha : U \to \mathsf{F}_1(U), u_0)$, (φ, id_U) is the cocartesian arrow for φ and p. Therefore, G is a cofibration. □

Often a service is specified via a collection of actions over a shared state space, each of which is specified by a contract and can be taken as an independent service. We can pack such different services together and get an aggregated service, which has an additive interface. Furthermore, its behavior is unique and already known from that of its component services.

Definition 4.3 For two independent services with different interfaces, but over the same state space U and initial state u_0

$$p_1 = (U, \alpha_1 : U \to A_1 \times \mathsf{B}_1(U \times O_1)^{S_1}, u_0)$$

and

$$p_2 = (U, \alpha_2 : U \to A_2 \times \mathsf{B}_2(U \times O_2)^{S_2}, u_0)$$

The aggregated service $p = p_1 \oplus p_2$ is defined by

$$p = (U, \alpha : U \to A \times \mathsf{B}(U \times O)^S, u_0)$$

where the observer of p is $o = \langle o_1, o_2 \rangle$, the action of p arises as the currying of

$$m = [\mathsf{B}_1(\mathsf{id} \times \iota_1) \circ m_1, \mathsf{B}_2(\mathsf{id} \times \iota_2) \circ m_2] \circ \mathsf{dr}$$

Theorem 4.4 *Service aggregation satisfies*[4] *:*

- $p \oplus p \sim p$.
- $p_1 \oplus p_2 \sim p_2 \oplus p_1$.
- $(p_1 \oplus p_2) \oplus p_3 \sim p_1 \oplus (p_2 \oplus p_3)$.

5 Comparing Services

When comparing services, one intuitively identifies models which, being non isomorphic at the data level, behave in a similar way "as far as we can see". Furthermore this tends to be the key ingredient in specifications of distributed systems whose "observational contents" (or parts thereof) are shared by different observers.

In [15], the notion of *bisimulation* was introduced in process algebra to capture this kind of observational equivalence between processes. Two processes are bisimilar to each other if there exists a bisimulation relationship between them, indicating how one process can be simulated by the other and vice versa. We also consider bisimulation a fundamental notion in service technology, as it seems to capture appropriately the "*black-box*" characterization of services.

A categorical definition with respect to coalgebras is given by Aczel and Mendler [1] as a relation $R \subseteq U \times V$ for two F-coalgebras (U, α) and (V, β) such that there is a F-coalgebra (R, γ) satisfying

$$\begin{aligned} \mathsf{F}(\pi_1) \circ \gamma &= \alpha \circ \pi_1 \\ \mathsf{F}(\pi_2) \circ \gamma &= \beta \circ \pi_2 \end{aligned}$$

from which we can get

$$\begin{aligned} & \langle \mathsf{F}(\pi_1), \mathsf{F}(\pi_2) \rangle \circ \gamma = \langle \alpha \circ \pi_1, \beta \circ \pi_2 \rangle \\ \equiv & \\ & \mathsf{F}(\langle \pi_1, \pi_2 \rangle) \circ \gamma = (\alpha \times \beta) \circ \langle \pi_1, \pi_2 \rangle \\ \equiv & \\ & \mathsf{F}(id_R) \circ \gamma = (\alpha \times \beta) \circ id_R \\ \equiv & \\ & \gamma = \alpha \times \beta \\ \equiv & \\ & (u, v) \in R \Leftrightarrow u \in U \wedge v \in V \end{aligned}$$

which means that any relation preserving the transition structures of α and β is a bisimulation between them.

The following diagram is the corresponding instantiation for the functor F as

[4] Note that the terms are not equal but bisimular to each other in the theorem. The formal definition of bisimularity will be given in Section 5.

given in (2) underlying our model of services.

$$\begin{array}{ccccc} U & \xleftarrow{\pi_1} & R & \xrightarrow{\pi_2} & V \\ \downarrow{\scriptstyle\alpha} & & \downarrow{\scriptstyle\gamma} & & \downarrow{\scriptstyle\beta} \\ A\times \mathsf{B}(U\times O)^S & \xleftarrow[A\times\mathsf{B}(\pi_1\times O)^S]{} & A\times \mathsf{B}(R\times O)^S & \xrightarrow[A\times\mathsf{B}(\pi_2\times O)^S]{} & A\times \mathsf{B}(V\times O)^S \end{array}$$

Then, let $\alpha = \langle o_\alpha, m_\alpha\rangle$, $\beta = \langle o_\beta, m_\beta\rangle$, $\gamma = \langle o_\gamma, m_\gamma\rangle$, a simple calculation yields that

$$\begin{aligned} &\alpha\circ\pi_1 \\ =&(A\times \mathsf{B}(\pi_1\times O)^S)\circ\langle o_\gamma, \overline{m_\gamma}\rangle \\ =&\langle o_\gamma, \mathsf{B}(\pi_1\times O)^S\circ\overline{m_\gamma}\rangle \\ =&\langle o_\gamma, \overline{\mathsf{B}(\pi_1\times O)\circ m_\gamma}\rangle \end{aligned}$$

and, similarily, $\beta\circ\pi_2 = \langle o_\gamma, \overline{\mathsf{B}(\pi_2\times O)\circ m_\gamma}\rangle$. A direct consequence of these equalities is the fact that, for any $\langle u, v\rangle \in U\times V$, the following equations hold:

$$\begin{aligned} &o_\gamma\langle u,v\rangle = o_\alpha u = o_\beta v \\ &\overline{\mathsf{B}(\pi_1\times O)\circ m_\gamma}\langle u,v\rangle = \overline{m_\alpha}u \\ &\overline{\mathsf{B}(\pi_2\times O)\circ m_\gamma}\langle u,v\rangle = \overline{m_\beta}v \end{aligned}$$

We may rephrase such results as a proof rule for bisimulation, whose shape depends on the adopted behavior monad B. For example, for the nondeterministic case $\mathsf{B} = \mathcal{P}$, the proof rule resembles the definition of bisimulation for classical labelled transition systems.

$$\begin{aligned} \langle u,v\rangle \in R \iff & o_\alpha u = o_\beta v \\ & \wedge \forall s\in S \\ & (\ \forall\langle u',t\rangle \in m_\alpha\langle u,s\rangle\ .\ \exists\langle v',t\rangle\in m_\beta\langle v,s\rangle\ .\ \langle u',v'\rangle\in R \\ & \quad\wedge \\ & \ \forall\langle v',t\rangle \in m_\beta\langle v,s\rangle\ .\ \exists\langle u',t\rangle\in m_\alpha\langle u,s\rangle\ .\ \langle u',v'\rangle\in R) \end{aligned}$$

Definition 5.1 Let $p = (U, \alpha : U \to \mathsf{F}U, u_0)$ and $q = (V, \beta : V \to \mathsf{F}V, v_0)$ be services over the same interface I. They are said to be bisimilar, written $p \sim q$, iff there is a F-bisimulation $R \subseteq U\times V$ containing the pair $\langle u_0, v_0\rangle$.

There is a close relationship between coalgebra homomorphisms and bisimulations: A map $h : U \to V$ is a homomorphism between (U,α) and (V,β) iff its graph is a bisimulation between (U,α) and (V,β). Is there any corresponding results for service morphisms? This question leads to the following definition of bisimulation up to ϕ.

Definition 5.2 Let $p = (U, \alpha : U \to \mathsf{F}U, u_0)$ and $q = (V, \beta : V \to \mathsf{F}'V, v_0)$ be services over interface I and I' respectively, and $\phi : \mathsf{F} \to \mathsf{F}'$ be a natural transformation

as given in Definition 4.1. Then p and q are said to be bisimilar up to ϕ, written $p \sim_\phi q$, iff $(U, \phi_U \circ \alpha, u_0) \sim q$.

6 Refinement

It is clear that if we want to develop applications of any size, we must be able to decompose their description into different services and compose the application from the developed services. This is just as true when the description is a contract as it is when it is a service.

In general, the construction of services is a process which involves a systematic trajectory for transformation, starting from "abstract" descriptions, leading to "concrete" ones. Refinement can be defined, in broad terms, as such a transformation. It changes the representation of a service, entailing a notion of substitution, but not necessarily equivalence.

In this section, we investigate two levels of refinement relations. We start by introduce the refinement of contracts, and then the refinement of services is discussed.

6.1 Contract Refinement

In general, a refinement of contracts involves two contracts: an abstract one and a concrete one as its refinement. The idea involved in contract refinement is that the concrete contract adds implementation details which are left open in the abstract contract. For example, it can reduce the level of underspecification / nondeterminism.

Contract refinement requires validity of the properties in the abstract contract after transformed into the concrete one. That is, if contract Ctr_2 is a refinement of Ctr_1, we need to know whether Ctr_2 is a "correct" refinement of Ctr_1. We say that Ctr_2 is correct if it meets the following two requirements:

- property preservation: All properties that can be proved about Ctr_1 can also be proved for Ctr_2 (but not in general vice versa);
- substitutivity: A component of Ctr_1 in a system can be replaced by a component of Ctr_2 and the resulting new system should implement all the functionalities of the earlier system.

Keeping this idea in mind, we can get the following definition describing contract refinement:

Definition 6.1 Let $Ctr_1 = (I_1, \mathbf{L}_1, \mathbf{A}_1)$ and $Ctr_2 = (I_2, \mathbf{L}_2, \mathbf{A}_2)$ be two contracts, we say that Ctr_2 is a refinement of Ctr_1, denoted by $Ctr_2 \sqsubseteq Ctr_1$ if there is an interface morphism $\phi : I_2 \to I_1$, such that for every component $p = (U, \alpha, u_0)$ as an implementation of Ctr_2, $p' = (U, \phi_U \circ \alpha, u_0)$ is an implementation of Ctr_1.

Theorem 6.2 *Contract refinement satisfies:*

- $Ctr \sqsubseteq Ctr$.
- *If* $Ctr_2 \sqsubseteq Ctr_1$ *and* $Ctr_3 \sqsubseteq Ctr_2$, *then* $Ctr_3 \sqsubseteq Ctr_1$.

- *If* $Ctr_2 \sqsubseteq Ctr_1$*, then for any* Ctr*,* $Ctr_2 \parallel Ctr \sqsubseteq Ctr_1 \parallel Ctr$*.*

6.2 Service Refinement

The basic idea of service refinement, which is called the *principle of substitutivity*, is rather simple: Intuitively, it is acceptable to replace one service by another, provided it is impossible for a user of the services to observe that the substitution has taken place. If a service can be acceptably substituted by another, then the second service is called a *refinement* of the first one.

There is a diversity of ways of understanding both what *substitution* means, and what such a *transformation* should seek for. For services, refinement can be addressed at two different levels (at least), namely, *behavioural* and *architectural refinement*.

- The *behavioral refinement* typically relates services of the same interface, where the refinement is based on a simulation preorder between the two services. Since morphisms between services of the same interface are in fact coalgebra homomorphisms which, therefore, entail bisimilarity, we built a weaker notion of a morphism between services, which still preserve the source service dynamics.
- The *architectural refinement* is used for decomposing a service with a specified behavior into a distributed architecture, i.e., a family of services being combined together, which is also modelled as a concrete coalgebra. The refined service is also a "behavioral refinement" of the given service with respect to the interface of the given service.

An order $\leq$ on a **Set** endofunctor F is defined in [9] as a functor $\leq$ (concretely, mapping every set U into a collection of preorders $(\mathsf{F}U, \leq_{\mathsf{F}U})$). In the sequel $\leq$ will be referred to as a *refinement preorder*.

Definition 6.3 Let F be an extended polynomial functor on **Set** and consider two F-coalgebras $p_1 = (U_1, \alpha_1 : W_1 \to \mathsf{F}U_1)$ and $p_2 = (U_2, \alpha_2 : U_2 \to \mathsf{F}U_2)$. A *forward* morphism $h : p_1 \to p_2$ with respect to a refinement preorder $\leq$, is a function from U_1 to U_2 such that

$$\mathsf{F}\, h \circ \alpha_1 \;\leq\; \alpha_2 \circ h$$

Dually, h is called a *backwards* morphism if

$$\alpha_2 \circ h \;\leq\; \mathsf{F}\, h \circ \alpha_1$$

Definition 6.4 Given services p and q, p is a behavioural refinement of q, written $p \sqsubseteq q$, if there exist services r and s such that $p \sim r$, $q \sim s$ and $r \sqsubseteq_F s$, where $r \sqsubseteq_F s$ stands for the existence of a (seed preserving) forward morphism from r to s.

In the case of large-scale applications consisting of many services, it is not practical to consider the whole system each time we want to refine one of its services. On the contrary, we prefer to do the refinement steps locally for the particular service being considered. Fortunately, behavioural refinement is well behaved in this respect.

Behavioral refinement characterizes what it means to preserve service behavior. But if our framework is based on behavioral refinement alone, the inability to change the syntactic interface will force us to work at the same level of interface abstraction throughout the whole development process. As a result, the development will be unnecessarily complex and inflexible. Furthermore, when we develop an application, it is certainly not enough to characterize its black-box behavior only, we also need to capture its internal structural aspects. For these purposes, we introduce the notion of *architectural refinement* in [14]. All the refinement steps in an architectural refinement are required to respect the architectural structure, and preserve the application behavior. Thus, it is a generalization of behavioral refinement.

Due to the length limitation, we only briefly introduce the notion of hehavioural refinement in this paper. More details about refinement for coalgebras can be found in [14]. Furthermore, taking QoS into consideration, we need to define a wider range of behavior. For example, an essential aspect of behavior of services is the execution time of certain (sequence of) operations. In other words, refinement of services should preserve all kinds of temporal constraints. We will not discuss such problems in this paper and leave them as future work.

7 Composing Services

In this section, we move on to a brief introduction of service composition. Composition enables prefabricated services to be reused by rearranging them in ever-new composites. Resulting composite services can be used as basic services in further compositions or offered as complete applications and solutions to service clients. Orthogonal to the *vertical* refinement of concrete services from abstract specifications, composition operations are needed to support the *horizontal* decomposition of the applications into component services. That means, an application is represented by specifying its component services and their composition. By stating how services are composed together, we can get a view of the architecture of the application.

In [4,13] two different frameworks for defining the combinators of coalgebras are investigated, and [5,3] provides a summarization for these two approaches. In this section, we will have a brief overview of the operators using the heterogeneous framework adopted by [13] for composition of services.

One widely used combinator for composing services is sequential composition, which connects the output interface of one service to the input interface of another, if the interfaces are declared to be compatible. This operator is formally defined as follows.

Definition 7.1 Let $p = (U_p, \alpha_p : U_p \to A_p \times \mathsf{B}_p(U_p \times K)^I, u_p \in U_p)$ and $q = (U_q, \alpha_q : U_q \to A_q \times \mathsf{B}_q(U_q \times O)^K, u_q \in U_q)$ be two services. Their sequential composition is formed by placing them side by side and connecting the output of p

to the input of q [5]. Formally, the sequential composition of p and q is given by

$$p;q = (U, \alpha_{p;q} : U \to A \times \mathsf{B}(U \times O)^I, \langle u_p, u_q \rangle \in U)$$

where $U = U_p \times U_q$, $A = A_p \times A_q$, $\mathsf{B} = \mathsf{B}_p\mathsf{B}_q$ [6]. And the dynamics $\alpha_{p;q}$ is represented by $o_{p;q} : U \to A$ and $a_{p;q} : U \times I \to \mathsf{B}(U \times O)$. The detailed definition is given as follows:

$$o_{p;q} : U_p \times U_q \xrightarrow{\langle o_p, o_q \rangle} A_p \times A_q$$

and

$$\begin{aligned} a_{p;q} : (U_p \times U_q) \times I &\xrightarrow{\mathsf{xr}} U_p \times I \times U_q \xrightarrow{a_p \times \mathsf{id}} \mathsf{B}_p(U_p \times K) \times U_q \\ &\xrightarrow{\tau_r^{\mathsf{B}_p}} \mathsf{B}_p(U_p \times K \times U_q) \xrightarrow{\mathsf{B}_p(\mathsf{a} \cdot \mathsf{xr})} \mathsf{B}_p(U_p \times (U_q \times K)) \\ &\xrightarrow{\mathsf{B}_p(\mathsf{id} \times a_q)} \mathsf{B}_p(U_p \times \mathsf{B}_q(U_q \times O)) \xrightarrow{\mathsf{B}_p\tau_l^{\mathsf{B}_q}} \mathsf{B}_p(\mathsf{B}_q(U_p \times (U_q \times O))) \\ &\xrightarrow{\mathsf{B}_p\mathsf{B}_q\mathsf{a}^\circ} \mathsf{B}_p\mathsf{B}_q(U_p \times U_q \times O) \end{aligned}$$

where a and s are two isomorphisms representing associativity and commutativity respectively:

$$\begin{aligned} \mathsf{a} &: A \times B \times C \to A \times (B \times C) \\ \mathsf{s} &: A \times B \to B \times A \end{aligned}$$

a° is the inverse of a, and $\mathsf{xr} = \mathsf{a}^\circ \circ (\mathsf{id} \times \mathsf{s}) \circ \mathsf{a}$ is the exchange morphism which changes the position of factors in the multiplicative expression:

$$\mathsf{xr} : A \times B \times C \to A \times C \times B$$

Services can be aggregated in a number of different ways, besides the 'pipeline' style modelled by the sequential composition. In [13], a family of operators have been defined. The restriction $\restriction$ and relabelling $-[\gamma]$ shows the possibilities of changing interfaces of services. Given two services p and q satisfying specified conditions, their external choice $p \boxplus q$, parallel composition $p \boxtimes q$ and concurrent composition $p \circledast q$ are defined separately by exploiting the universal constructions in the category **Cop**. When interacting with $p \boxplus q$, the environment will be allowed to choose either to input a value of input type of p or that of q, which will invoke the corresponding service (p or q, respectively), producing the relevant output. On its turn, *parallel* composition corresponds to a synchronous product: both services are executed simultaneously when triggered by a pair of legal input values. Note, however, that the behavior effect, captured by monad B, propagates. For example,

[5] In general, the output type K of p does not need to be same with that of the input channel of q, which we denote by L. The condition that $K \subseteq L$ is enough. But for simplicity, we still let $L = K$ in the following definition.

[6] The simple composition of the corresponding functors B_p and B_q does not always lead to new monads. In order to define a new monad based on $\mathsf{B}_p\mathsf{B}_q$ for two given monads B_p and B_q, a natural transformation $\lambda : \mathsf{B}_q\mathsf{B}_p \to \mathsf{B}_p\mathsf{B}_q$ satisfying a number of conditions should exist. The more detailed result on this topic can be found in [6].

if B can express behavioral failure and one of the arguments fails, the product will fail as well. Finally, *concurrent* composition combines choice and parallel, in the sense that p and q can be executed independently or jointly, depending on the input supplied. Generalized interaction is catered through a sort of 'feedback' mechanism on a subset of the input ends, which can be defined similarly as that in [4].

8 Concluding Remarks

In this paper, we have shown how a formal framework for assisting service-oriented applications development can be defined around the notions of interface, contract, service and corresponding notions of morphisms. In such a formalism, services are described by coalgebras whose signature functors are derived from the corresponding interfaces, and satisfy the axioms given in corresponding contracts. This work provides a unifying framework for different notions (interfaces, contracts and services) in SOA development.

With respect to the composition of services, we adopt the categorical approach which can be traced back to the work of Goguen *et al*[8]. Indeed, the category of services forms the cofibration over the corresponding category of interfaces, and thus provides the starting point for the definition of combinators for services. Preliminary work can be found in [13].

Concerning locality of services in the whole application, a family of operators for composing services are introduced and a framework for building application by aggregating services is provided. This allows developers to decompose an application into services, discover different services separately and finally build the system by composing the services.

One immediate topic for future work is to investigate the composability of services for replaceability, compatibility and conformance. Another research challenge is to integrate QoS aspects of services [12] into the coalgebraic model and build a QoS-aware calculus for composing services.

References

[1] P. Aczel and N. Mendler. A final coalgebra theorem. In D.H. Pitt, D.E. Rydeheard, P. Dybjer, A. Pitts, and A. Poigne, editors, *Proceedings of Category Theory and Computer Science*, volume 389 of *LNCS*, pages 357–365. Springer, 1989.

[2] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business process execution language for web services version 1.1, 2003. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/.

[3] Luís S. Barbosa, Sun Meng, Bernhard K. Aichernig, and Nuno Rodrigues. On the semantics of componentware: a coalgebraic perspective. In Jifeng He and Zhiming Liu, editors, *Mathematical Frameworks for Component Software - Models for Analysis and Synthesis*. World Scientific, 2006.

[4] Luís Soares Barbosa. *Components as Coalgebras*. PhD thesis, Universidade do Minho, Braga, Portugal, 2001.

[5] Luís Soares Barbosa and Sun Meng. Generic components. In Graham Hutton, editor, *Proceedings of First APPSEM-II Workshop*, Nottingham, March 2003. APPSEM Network Report.

[6] Michael Barr and Charles Wells. *Toposes, Triples and Theories*. Springer, 1985.

[7] José Luiz Fiadeiro and Tom Maibaum. Categorical Semantics of Parallel Program Design. *Science of Computer Programming*, 28:111–138, 1997.

[8] Joseph Goguen. Categorical Foundations for General Systems Theory. In F. Pichler and R. Trappl, editor, *Advances in Cybernetics and Systems Research*, pages 121–130. Transcripta Books, 1973.

[9] Bart Jacobs and Jesse Hughes. Simulations in coalgebra. In H. Peter Gumm, editor, *Elect. Notes in Theor. Comp. Sci. (CMCS'03 - Workshop on Coalgebraic Methods in Computer Science)*, volume 82, pages 245–263, Warsaw, April 2003.

[10] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of the European Association for Theoretical Computer Science*, 62:222–259, 1997.

[11] M. P. Papazoglou and D. Georgakopoulos. Service Oriented Computing. *Comm. ACM*, 46(10):25–28, 2003.

[12] Sun Meng. QCCS: A Formal Model to Enforce QoS Requirements in Service Composition. In J. He and J. Sanders, editors, *Proceedings of 1st IEEE & IFIP International Symposium on Theoretical Aspects of Software Engineering, TASE'07*, pages 389–400. IEEE Computer Society, 2007.

[13] Sun Meng and Bernhard Aichernig. A Coalgebraic Calculus for Component-based Systems. In Hung Dang Van and Zhiming Liu, editor, *Proceedings of the Forkshop on Formal Aspects of Component Sofsware FACS'03*, pages 27–46, 2003.

[14] Sun Meng, Luís S. Barbosa, and Zhang Naixiao. On Refinement of Software Architectures. In *Proceedings of ICTAC'05*, volume 3722 of *LNCS*, pages 482–497. Springer, 2005.

[15] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[16] Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-Oriented Computing: State of the Art and Research Challenges. *IEEE Computer*, pages 64–71, 2007.

[17] Jan Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249:3–80, 2000.

[18] Clemens Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[19] Universal Description, Discovery, and Integration (UDDI) v3.0. http://www.uddi.org/.

[20] W3C. Simple object access protocol (soap) v1.2. http://www.w3.org/2000/xp/Group/.

[21] W3C. Web Service Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl.