ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/221610929

On systematic Design of Service Oriented Architectures.

Conference Paper · January 2008

Source: DBLP

CITATIONS
2

READS
59

3 authors:

Benjamin Kanagwa
Makerere University
36 PUBLICATIONS **203** CITATIONS

SEE PROFILE

Ezra Mugisa
45 PUBLICATIONS **125** CITATIONS

SEE PROFILE

Theo P. van der Weide
Radboud University
269 PUBLICATIONS **2,822** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

PhD project View project

TREC Reports View project

All content following this page was uploaded by Theo P. van der Weide on 26 December 2013.

The user has requested enhancement of the downloaded file.

On Systematic Design of Service-Oriented Architectures

Benjamin Kanagwa
Radboud University Nijmegen
The Netherlands
Email: B.Kanagwa@cs.ru.nl

Ezra K Mugisa
University of The West Indies
Kingston, Jamaica
Email: ezra.mugisa@uwimona.edu.jm

Th.P. van der Weide
Radboud University Nijmegen
The Netherlands
Email: tvdw@cs.ru.nl

Abstract— **We present a systematic means of designing service-oriented systems. Because services are developed independently, with no prior knowledge about each other, there is only a limited possibility that such services use similar message templates, initiate calls to any other service, or generate messages to support desired architecture configurations. As a result, construction of service-oriented systems rely on *intermediary services* to pre-process, transform and route messages to appropriate locations. The paper suggests a systematic design for the intermediary services and construction of services-oriented systems using a reuse drive approach. The approach is founded on category theory- a formal foundation for capturing and preserving structures. The approach supports *parallel* and *incremental inter-connection* of services in a planned and reusable manner.**

I. Introduction

Service-Oriented Systems are constructed from independent services that are composed and coordinated to satisfy a set of functional and non functional requirements. The consumer may be an end-user or another service. In service-to-consumer interactions, the behavior of the provider is predefined and any adjustments are possibly on the side of the consumer. The consumer can interact in any way that does not violate the rules set by the provider. The key question we aim at answering is how to transform *independent* services into a service-oriented system in a systematic manner. By systematic we mean a way that is reusable based on a set of principles and guidelines that provide a sound step-by-step methodology for putting service-oriented systems together.

In this paper, we propose a design approach that blends the structure of services, the composition, coordination, the design and placement of intermediary services into a single formal framework that imposes a sound incremental methodology. Our idea is to rely on category theory [1], [2] which inherently preserves structure and provides tools for composition and incremental construction. Most previous approaches based on Finite State Machines [3], Petrinets [4], Process Algebra [5] emphasize description of service compositions rather than design and construction.

Construction of service-oriented systems is a function of composition and coordination of services. Because services combine on the basis of what they offer, the process of construction of service-oriented systems is the process of consuming services [3]. The importance of composition of services is reflected by the current interest in the subject with most recent efforts targeting automatic composition [6], [7], [8], [9]. Business Process and Execution language for Web Services (BPEL4WS) [10], now a standard for description of composition of services allows one to describe a business process in terms of the behavior of other services. It provides for coordination of services based on the actions. It assumes a single output message for each action. BPEL4WS does not offer a construction approach but a description framework. However, because services are independently designed, owned and deployed, it is a rare occurrence that the consumption of services is free of challenges. Challenges stem from lack of compatibility both at the syntactic and semantic level. Even when they are compatible, there is usually need for intermediary services to ensure smooth coordination of services and address any additional concerns that may be necessary to fulfill the goal of the service-oriented system.

Focusing on service intermediaries is reasonable because they have important responsibilities that include adaptation, coordination and general routine keeping such as logging. Our strategy is to make the coordination of services together with design and placement of service intermediaries more systematic and principled. Our intention is to provide systematic support for direct messaging, and construction of intermediary services that optionally involve the consumer.

A. Web Service SOAP Solution

Web services [11] rely on Simple Object Access Protocol(SOAP) [12] as the messaging framework. Within the SOAP framework, direct messaging is supported through acyclic message processing as the SOAP envelop moves from node to node. Below we highlight the solution suggested by the web services community.

The SOAP [12] distributed processing model assumes that a SOAP message originates at an initial SOAP sender and is sent to an ultimate SOAP receiver via zero or more SOAP intermediaries. However, SOAP itself does not define any routing or forwarding semantics. It is anticipated that such functionality can be described as one or more features and expressed as SOAP extensions or as part of the underlying protocol binding. A *SOAP intermediary* is a node which forwards a message to another SOAP node on behalf of the initiator of the inbound SOAP message. SOAP defines two

different types of intermediaries: *forwarding* intermediaries and *active* intermediaries.

Forwarding intermediaries process the message according to only the rules specified in the SOAP message, while the *active* intermediaries may undertake additional processing beyond what is in the rules in the SOAP message. According to the SOAP specification, it is expected that features provided by *active* intermediaries are described in a manner that allows such modifications to be detected by the affected SOAP nodes. For example, an *active* intermediary may describe the processing performed by inserting header blocks into the outbound SOAP message that inform downstream SOAP nodes acting in roles whose correct operation depends on receiving such notification. The semantics of such inserted headers require that either the same or other headers to be (re)inserted at subsequent intermediaries as necessary to ensure that the message can be safely processed by nodes yet further downstream. For example, if a message with headers removed for encryption passes through a second intermediary (without the original headers being decrypted and reconstructed), then indication that the encryption has occurred must be retained in the second relayed message.

According to this specification, it is risky for a service integrator to rely on *active* intermediary nodes that are not under his/her control. The likely situation is that active intermediary nodes may turn out to be *coordination points* designed and controlled by the service integrator. Most of the work of a service-oriented engineer is to design and manage the coordination points.

During composition, the way messages synchronize is non trivial because some messages become internal to the composite service [13] while new messages may have to be introduced to complete the interaction. The way messages become internal and those that become exposed by the composite service is fundamental to the composition of services. Service composition is a recursive problem because composite services can be exposed as new 'atomic' services whose coordinated behavior is concealed from the consumer. With this in mind, construction of service-oriented systems can be carried out in an *incremental* manner.

II. Example and Practical Scenario

We use the example from Web services Interoperability Organization [14], [15] with sequence diagrams in Fig.1. The application being modeled is that of a Retailer offering Consumer electronic goods to Consumers; a typical B2C model. To fulfill orders the Retailer has to manage stock levels in warehouses. When an item in stock falls below a certain threshold, the Retailer must restock the item from the relevant Manufacturer's inventory (a typical B2B model). In order to fulfill a Retailer's request a Manufacturer may have to execute a production run to build the finished goods. The above scenario involves three simple services, the Shop, Warehouse and Manufacturer. Since, we can have many instances of each service; the example is rich enough to demonstrate our formal systematic design. Starting with these three services, we want to explain how to compose them *directly* without implying changes in the original services. While composing services in a service-oriented environment, the following restrictions apply to the service-oriented approach *(i)* services are consumed *'as is'* and therefore we can not change the templates of the messages they send or receive. *(ii)* services operate in a *request–response/client–server* manner. So a reply is always sent to the requestor of the service.

III. The approach

Our work extends earlier work, including some parts that are most closely related to work in [16], [17], by allowing explicit separation of guarantees from outputs and creation of classes over action outputs. We also explicitly model replication of outputs which allows a larger set of architecture configurations. Our approach is also constructive; one can construct a larger system by following a step-by-step methodology. The approach is based on the notions of *coordinated activity, guarantees (semantic implication)* and *public view* of services. The idea of guarantee is informally introduced in [3], [16]. It can also be related to triggers in [18]. Consider the *shop* service in Fig. 1, which specifies actions *getCatalog* and *submitOrder*. Ideally these actions combine to perform the *activity sellItem* which could result in an item being *SOLD* or *FAIL* for some reason. *SellItem* is not considered as an action but rather a coordinated activity because (*i*) it is not *transparently* presented to the user (*ii*) it is entirely in terms of other actions. By transparency we mean that a user does not invoke *sellItem* without being directly involved in other intermediary actions. The actions *getCatalog* and *submitOrder* make the *public view* of the shop service, because they are directly invoked by the consumer.

Normally, the expected results of a coordinated activity do not match (both in *number* and *type*) the outputs of the *actions* involved and this creates a source of behavior incompatibility [19], [20]. To mitigate this challenge, we create 'classes' of coordination over the outputs of actions. The outputs of an action are partitioned into sets that associate meaning in terms of the coordinated activity.

The rest of the approach is cast in a categorical setting that preserves structure, presents a composition and coordination framework. The specific contributions of this paper are (i) structure for representing services and service-oriented systems. The structure is formal which implies there is a precise and concise meaning associated to it. (ii) We present a systematic and incremental means of constructing larger systems based on a categorical approach making the process more planned and reuse driven (iii) We provide a criteria for design and placement of service intermediaries to complete the process of service-oriented design. (iv) Finally we present loose semantics for the core building blocks of service intermediaries.

One advantage with our approach is that it preserves the structure of services, which makes the process of constructing services incrementally reused and systematic. This approach provides for intermediate services (wrappers) that perform the

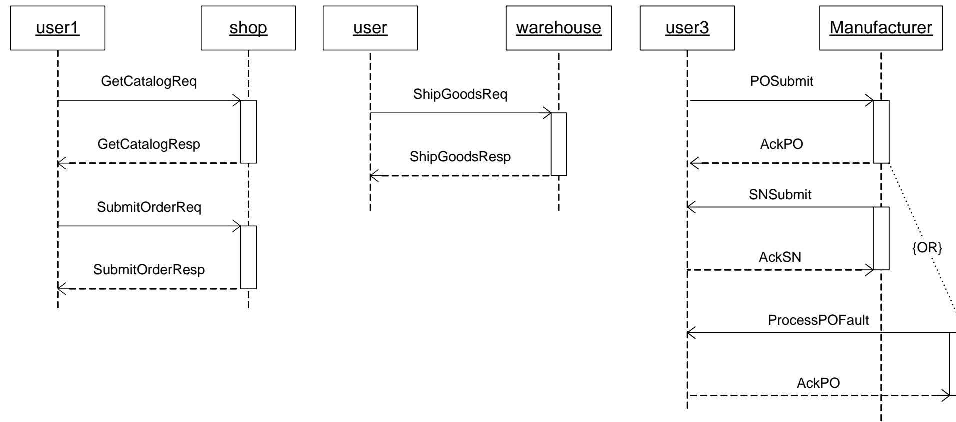

Fig. 1. Sequence diagrams for Shop, Warehouse and Manufacture services(Drawn initially as autonomous services with no assumption of prior knowledge between the services)

role of transforming, coupling messages and adaptation of services.

IV. RELATED WORK

In [18], [21], a category theoretical approach to service-oriented modeling is presented. Their categorical formalization is centered on the notion "coordination contract" which they relate to their earlier work on software architectures, Parallel Program Design and Reconfigurable distributed systems all coached in different categorical settings. As the name suggests, coordination interfaces are intended to coordinate systems. Algebraic approaches proposed by [22] aim to capture coordination in the co-span object. Our work is directed towards design with emphasis on intermediaries and structure of services.

In [16], a set of interface theories for web services are presented. The work emphasizes the interface theory with models for substitutability, compatibility and well-formedness of interfaces. Well-formedness is also discussed in [3]. We include these concerns through our concept of *classes* over action outputs.

Other related work deals with issues of automatic composition [8], [3], [9], [23] where the target is to find a set services and combine them to fulfill a request. A request may be just a result/output or a service that exhibits a given behavior. Although the major aim there is formal description of automatic composition approaches, the models used form a major part of construction and computational structures in our categorical setting especially tree-based structures.

In [6], [24] they deal with adaptation, replaceability and compatibility. In relation to these concepts, our approach is not expressive enough especially in terms of behavioral adaptability at the level of service behavior.

V. THE CATEGORICAL APPROACH

The approach proposed here is guided by three principles. First, the process of constructing a service-oriented system is the process of consuming services [3]. Second, in agreement with [25], we want to keep the concept of service at the center of the construction (design) process. So, we would like to keep a clear view of the consumer and provider entities in the entire process. Third, we would like to support the autonomy and reactive nature of services as much as we can. Our key guiding factor here is that services combine on the basis of what they provide independently but not on the basis of what they offer to each other. For instance, a *medical* service provider can combine with an *insurance* service provider not because the two providers (directly) offer anything to each other, but because they produce an added valued of a *medical insurance* service when combined.

To support these concepts into the design of service-oriented systems in a systematic manner, we use category theory [1]. A *c*ategory E consists of two classes, the members of the first of which denoted by the letters $X, Y, \ldots$ – are called *objects* (*structures*) and the members of the second of which denoted by the letters $f, g, \ldots$ – are called *arrows* (*morphisms*). Each arrow f is assigned an object X as *domain* and an object Y as *codomain*, indicated by writing $f : X \to Y$. If g is any arrow $g : Y \to Z$ with domain Y, the codomain of f, there is an arrow $fg : X \to Z$ called the composition of f and g. For each object Y there is an arrow $id_Y : Y \to Y$ called the identity arrow of Y. These notions are assumed to satisfy the following identity and associativity axioms:

$f.id_Y = f, id_Y.g = g, f(gh) = (fg)h$ for any arrows $f : X \to Y, g : Y \to Z, h : Z \to W$

Given two categories D and E, a *f*unctor F from D to E consists of a pair of functions (both denoted by F),one from the class of objects of D to that of E, and the other from the class of arrows of D to that of E, such that

if $f : X \to Y$ in D , then $F(f) : F(X) \to F(Y)$ in E ; $F(id_X) = id_{F(X)}$, for all objects X of E
and

$F(gh) = F(g)F(h)$ for composeable arrows h, g of D

Morphisms preserve the structure of the objects. In software engineering morphisms $\rho : X_1 \to X_2$ in a suitable category are used to define (i) *r*efinement - specifies a way in which the target object is a refinement of the source object, (ii)

inclusion- capturing legal transformations on objects to form more complex objects. Inclusion morphisms specify part-of relation (iii) *interpretations* - capture relationships between theories and can be used to model relationships between abstract and concrete implementations (iv) *simulations* - target execution behavior of systems such as behavior of the stack.

With the categorical approach, suitable categories are used to represent the structure of different artifacts that make up the target entity such as a service or service-oriented system. The process of putting small services together into bigger services/systems is then based on standard categorical techniques such as *colimits* [26].

Our use of category theory is restricted to defining and preserving the structure of services and providing a systematic means of constructing larger services from smaller services in an incremental and principled manner through the *pushout* construction [2]. The morphisms that we target are therefore inclusion morphisms, that aim at expressing how smaller services become part of larger services.

VI. Notation For Representing Service Artifacts

This section presents the general notation for representing service artifacts such as messages and actions and their relationships. The primitive artifacts are actions and activities. Rules imposed on the relationship of these artifacts yield new artifacts such as service interfaces. From now on, M^I and M^O are the sets of input and output messages respectively and A, a set of actions.

A. The Actions

A service action is associated with a single input message and zero or more output messages. Each output has a guarantee associated with it. A guarantee is used to provide (i) evidence of occurrence of an action (ii) semantic interpretation to create 'classes' of coordination based on the activity in which an action participates. Let $\mathcal{O}(a)$ represent the set of possible outputs of action *a*. Consider the example of an action *submitOrder*, with input *SubmitOrderReq* and corresponding output and guarantee pairs; *submitOrder*(*SubmitOrderReq*) ={*<SubmitOrderResp, OK>*, *<BadOrder, Fail>*, *<InvalidCode, Fail>*, *<ConfigFault, Fault>*}. The model assumes that only a single output is possible at any given moment in time. However, the design of intermediary services allows for replication of outputs to coordinate more services using single output. Below is a formal definition of an action alphabets.

DEFINITION 1 The output alphabet of an action a is a pair $\langle M, F\rangle$, where F is a total map from M into $\mathcal{O}(a)$, where M is the set of actions that process message M. An action morphism for an action a in M, denoted by $f_M : A_1 \rightarrow A_2$ is a total map such that $F_1(a) = F_2(f_M(a))$ for all actions $a \in M$.

The action alphabet and action morphism form a category of actions *ACT*

The intention of the above definition is similar to idea in [27] where slot and slot morphisms are defined. The underlying intension is to synchronize services on the *type* of output for each action. Therefore each $\mathcal{O}(a)$ defines the codomain of action a and $F(a)$ is the actual message/value that is output by the action a. Consider an action *SubmitOrderReq*. The types of messages that it can output are seen as 'variables' that can be changed by the action. Viewed in this perspective the slots of different actions are disjoint. Put in another way, no two actions can change the same slots. In other words, the output types of each action are unique.

B. Service Activity

A service activity is coordinated over a set of *visible* actions. An action is visible if it can be invoked directly by a consumer. A visible action may invoke a series of other actions which return one of the expected outputs. Consider the activity *sellItem* with possible outputs *SOLD* or *FAILED*. An activity has no explicit inputs; they depend on the sequence of actions involved.

EXAMPLE 1 Below is an example of an activity

SellItem[SOLD, FAILED]
getCatalog:⟨<GetCatalogResp, submitOrder>,
<GetCatalogFailed, SellItem.FAILED>⟩
submitOrder:⟨<SubmitOrderResp, SellItem.SOLD>,
<BadOrder, SellItem.FAILED>,
<InvalidCode, SellItem.FAILED>,
<ConfigFault, SellItem.FAILED>⟩

In this activity, the *GetCatalog* action has two possible outputs. The first output, *GetCatalogResp* is coordinated with the action *submitOrder*. The second output *GetCatalogFailed* has been 'interpreted' to imply *FAILED* in terms of the coordinated activity. The idea behind the guarantees as used in this article is to partition all action outputs into classes of the guarantees of the coordinated activity. This technique facilitates adaptation because the coordinated actions are able to account for all the behavior (outputs) expected by the coordinated activity. Note that although this example is structurally similar to that used in [16], our approach is significantly different. First the coordination is based on both actions and the different outputs associated with them. Secondly, explicit separation of outputs and guarantee/semantic implications allows one to form classes over the outputs of actions - a technique that enforces adaptation of services to support coordinated activities. In this example, the outputs of each action are partitioned into two classes determined by the output of the coordinated activity. Because the activity *SellItem* is coordinated over actions, the *GetCatalogResp* output is followed by *submitOrder* which also determines its class/guarantee. The outputs of action *submitOrder* have the following classes: *SOLD:*{*SubmitOrderResp*}, *FAILED:*{*BadOrder, InvalidCode, ConfigFault*}. Although we have used outputs for clarity (because they have descriptive names), classes are formed over the corresponding *guarantee/semantic implication*.

action	input	output	guarantee
submitOrder	*SubmitOrderReq*	*SubmitOrderResp*	*OrderOk*
		BadOrder	*Fault*
		InvalidCode	*Fault*
		ConfigFault	*Fault*
getCatalog	*GetCatalogReq*	*GetCatalogResp*	*GetCatOk*

TABLE I
STATIC REPRESENTATION OF A SHOP SERVICE (ADAPTED FROM [15])

The classes/guarantee of outputs connected to other actions are determined by the final actions that they invoke. Since outputs of actions may be connected to other actions in different ways, there is need for a mechanism of evaluating guarantees. The connection can be sequential or in parallel. Parallel connections have two variations, one as alternative options and the other as complementary options. We use the symbols $\cap, \cup$ and $\uplus$ to represent sequential, parallel alternatives and parallel complements respectively. They are evaluated as follows

- $G1 \cap G2 = G2$ the guarantee of the last action
- $G1(G2 \cup G3) = G1$ if G2 or G3 imply G1 otherwise take G2 or G3
- $G1(G2 \uplus G3) = G1$ if both G2 and G3 imply G1 otherwise take G2 or G3.

VII. SERVICE STRUCTURE

The service structure consists of a collection of actions, input messages and output messages.

DEFINITION 2 A service signature Φ is defined as follows

$$\left| \begin{array}{ll} M^I & \text{:set of input messages} \\ M^O & \text{:set of output messages} \\ \text{A} & \text{:a set of actions} \end{array} \right.$$

EXAMPLE 2 The signature of the shop service can be represented as follows

$$\begin{array}{ll} M^I = & \{\textit{SubmitOrderReq, GetCatalogReq}\} \\ M^O = & \{\textit{SubmitOrderResp,BadOrder, InvalidCode,} \\ & \textit{ConfigFault, GetCatalogResp}\} \\ A = & \{\textit{getCatalog, submitOrder}\} \end{array}$$

DEFINITION 3 A service signature morphism $h = (M_1^I, M_1^O, A_1) \rightarrow (M_2^I, M_2^O, A_2)$ is defined as tuple $< h_I, h_O, h_A >$ defined as follows

$$\left| \begin{array}{ll} h_I & : M_1^I \rightarrow M_2^I \\ h_O & : M_1^O \rightarrow M_2^O \\ h_A & : A_1 \rightarrow A_2 \end{array} \right.$$

A. Service-Oriented System Structure

The notion of design used is based on the concept of coordinated activities. We perceive a service-oriented system as a connection of various services coordinated to perform a specific activity. A service-oriented system may be coordinated to perform several activities.

A *coordinated activity* has the following conditions

(i) It has set of possible outcomes. For example for the shop service we shall define an activity *sellItem* as having two possible classes of outcomes $\{$*SOLD, FAIL*$\}$.
(ii) An activity has a *public view* part and a detailed *coordinated part*. The public view of the activity is defined only in terms of actions that are directly accessible and invoked by the consumer.
(iii) For each action that takes part in any activity, its possible output is partitioned over the set of *outcomes* of an *activity*. The idea is for each output of an action, we need to know its implication in terms of the activity outcomes.
(iv) During incremental design, two actions from the same public view must not violate any order imposed by the service for that public view.

Condition (i), ensures that we separate between *coordinated activities* and actions defined by the service. The inputs of the activity are determined by the different inputs of the actions involved. Condition (ii), ensures that coordinated activities are transparent to the user. This is further enhanced by restricting that all activities map their outputs to those of the coordinated activity. Condition (iii), ensures that adaptability of service is built within the framework, by accounting for the behavior of actions in terms of the activities it coordinates.

The idea behind condition (iii) of partitioning guarantees of synchronized outputs over activity outputs is to provide a simple technique for establishing a ‘subtype’ relationship between an activity and actions that synchronize at different points. In a ‘subtype’ relation, to avoid unexpected outcomes, all behaviors (outputs and methods) of the subtype must be accounted for [28]. Therefore, the guarantee mapping between outputs of actions and outputs of coordinated activity ensures that any extra outputs by synchronized activities do not introduce any extra behavior that was not anticipated by the coordinated activity.

Condition (iv) is to ensure that if a service participates in the same activity more than once, it must ensure that new actions are inline with the relative ordering(specified by the instance) between the action and actions already involved. This minimizes potential deadlocks.

DEFINITION 4 A service-oriented system signature Ψ is defined as a pair (Φ, Δ) with morphisms $k = (\Phi_1, \Delta_1) \rightarrow$

(Φ_2, Δ_2) such that $h_\Phi : \Phi_1 \rightarrow \Phi_2, k_\Delta : \Delta_1 \rightarrow \Delta_2$ such that Φ is the service signature and Δ is a set of activities.

The morphisms in definition 4 are inclusion morphisms and they establish the classes of the coordinated activity as well the outputs on which services synchronize. The above structural representation is similar to that used in BPEL4WS [29] also used in [30]. Examples of service representation are given in the next section.

VIII. Complete Example Construction

Table II shows the actions and outputs of the warehouse *WH* and Manufacturer, *MF* whose specifications are given below

$$\begin{aligned} WH \equiv \quad M^I &= \{\text{ShipGoodsReq}\} \\ M^O &= \{\text{ShipGoodsResp, ConfigFault}\} \\ A &= \{\text{shipGoods}\} \\ \Delta &= \varnothing \end{aligned}$$

$$\begin{aligned} MF \equiv \quad M^I &= \{\text{POSubmit}\} \\ M^O &= \{\text{ackPO,POFault,ConfigFault}\} \\ A &= \{\text{submitPO}\} \\ \Delta &= \varnothing \end{aligned}$$

To combine the shop and warehouse in the coordinated activity *sellItem*, we use a shared object (*CH*) which identifies the actions, output and guarantees. Use of shared objects (cospan) is standard technique[31] to construct new combined objects through pushout construction.

$$\begin{aligned} CH_1 \equiv \quad M^I &= \{\text{u}\} \\ M^O &= \{\text{Sold,Failed}\} \\ A &= \{\text{a}\} \\ \Delta &= \varnothing \end{aligned}$$

The morphisms f and g

f : $u \mapsto$ *SubmitOrderReq*, *Sold* $\mapsto$ {*SubmitOrderResp*} *Failed* $\mapsto$ {*BadOrder, InvalidCode, ConfigFault*}, $a \mapsto$ {*submitOrder*},

g : $u \mapsto$ *ShipGoodsReq*, *Sold* $\mapsto$ {*ShipGoodsResp*}, *Failed* $\mapsto$ {*ConfigFault*}, $a \mapsto$ {*shipGoods*} . The corresponding categorical diagram is given below. The pushout for S' is given in Fig.2 (because of space, only the coordination part is presented) S'

Fig. 2. Categorical diagram : Shop, Ware house, Manufacturer

is the pushout construction of the shop *SH* and the warehouse *WH*, followed by an incremental composition of the MF which leads to S'', the the coordinated structure of *shop, warehouse* and *manufacturer*. The combined structure of S' is given in Fig. 2. The structure of S'' is similarly constructed using appropriate actions.

SellItem[SOLD, FAILED]
getCatalog:⟨<*getCatalogResp, submitOrder*>,
<*getCatalogFailed, SellItem.FAILED*>⟩
submitOrder:⟨<*SubmitOrderResp, shipGoods*>,
<*BadOrder, SellItem.FAILED*>,
<*InvalidCode, SellItem.FAILED*>,
<*ConfigFault, SellItem.FAILED*>⟩
shipGoods:⟨<*ShipGoodsResp, SellItem.SOLD*>,
<*ConfigFault, SellItem.FAILED*>⟩

Fig.2. Shop, Warehouse and Manufacturer Configuration

The structure is Fig.2 can be equivalently represented as a tree structure with actions as nodes and outputs as branches. A node with many branches is translated into an intermediary service.

IX. Intermediary Services and Systematic Design

We have already expressed the importance of intermediary services in the construction of service-oriented systems. Service intermediaries serve a role similar to adaptors and message brokers [32]. Adapters may intermediate between other applications rather than services. We limit our scope of service intermediaries to mediating between services.

We propose basic building blocks for the intermediaries which form most of the implementation for service-oriented systems that rely on third party services. We therefore think that once the construction of intermediaries is handled in a principled manner, then we have a systematic means of designing service-oriented systems. The categorical constructions above give a 'specification' of service intermediaries, which may be implemented as a single centrally coordinated service or split into multiple intermediaries.

A. Primitive Intermediary Services

Intermediary services serve both as *forwarding* and *processing nodes* in sense that they ensure that messages forwarded are in the format expected by the next service. Specifically, intermediary services pre-process messages and then route them to their destinations. In this design, an intermediary node may ask to receive the response or be sent to another intermediary service. The combined collaboration of the intermediary services together with other nodes (predefined services) define a service-oriented system architecture. We generalize three primitive intermediary services; *Forwarder*, *Replicator* and *Joiner* that can be used to build any architecture of a service-oriented system.

- *Forwarder*: receives a message from one node, pre-process it and forwards it to another node
- *Replicator*: receives a message from a single node, pre-process it into multiple messages that are sent to different nodes
- *Joiner*: receives a message from multiple sources, pre-process and then sends to a single node

We capture the above primitives in a systematic manner using the categorical framework. They can therefore be extended to any number of messages.

In the following subsections, we provide loose semantics for the *forwarder, replicator* and *joiner* intermediary services.

action	input	output	guarantee
WARE HOUSE SERVICE			
shipGoods	*ShipGoodsReq*	*ShipGoodsResp*	*Ok*
		ConfigFault	*Fault*
WARE HOUSE CALLBACK SERVICE			
submitSN	*SNSubmit*	*ackSN*	*orderOk*
		CallbackFault	*Fault*
		ConfigFault	*Fault*
errorPO	*ProcessPOFault*	*ackPO*	*submitPOFault*
		ConfigFault	*Fault*
		CallbackFault	*Fault*
MANUFACTURER SERVICE			
submitPO	*POSubmit*	*ackPO*	*Ok*
		POFault	*Fault*
		ConfigFault	*Fault*

TABLE II
WARE HOUSE, CALLBACK AND MANUFACTURER SERVICES

1) Forwarder Semantics: Below we give the general structure of a forwarding intermediary service.

$$\begin{aligned} FWD \equiv \quad Ports = \quad & \{in, out\} \\ M^I = \quad & \{\text{y}\} \\ M^O = \quad & \{\text{u}\} \\ \Delta = \quad & [m := in(y) \\ & u := transform(m) \\ & out(u)] \end{aligned}$$

An action can lead to different outcomes; we shall assign an output location (port) for each possible outcome associated with an input message. The transformation function represents the pre-processing needed by the service integrator such that there is syntactic compatibility with the providers to be interconnected. The transformation may involve renaming the message, pre-processing of message components or removal of message components to ensure syntactical compatibility with the next service provider.

2) Replicator Semantics: Below is the general structure of a replicating intermediary service.

$$\begin{aligned} RPL \equiv \quad Ports = \quad & \{in, out_1, ..., out_n\} \\ M^I = \quad & \{y\} \\ M^O = \quad & \{u_1, ..., u_n\} \\ \Delta = \quad & [\ m := in(y) \\ & for(i = 0\ to\ n) \\ & \quad do \\ & \quad u_i := transform(m) \\ & \quad out(u_i) \\ & od\ ] \end{aligned}$$

For a single input message, it transforms it into multiple messages to be sent to different service providers. The transformation may involve simple renaming of messages in which exact copies are reproduced or it may involve pre-processing of message components or removal of message components to ensure syntactical compatibility with the target service providers.

3) Joiner Semantics: This a general structure of a joining intermediary service.

$$\begin{aligned} JNR \equiv \quad Ports = \quad & \{in_1, ..., in_n, out\} \\ M^I = \quad & \{y_1, ..., y_n\} \\ M^O = \quad & \{\text{u}\} \\ \Delta = \quad & [\text{for(i = 0 to n)} \\ & \quad \text{do} \\ & \quad m_i := in_i(y_i) \\ & \quad \text{od} \\ & u := transform(m_1, ..., m_n) \\ & \text{out}(u)] \end{aligned}$$

As expected, a joiner intermediary service does the opposite of the replicator. It receives information from multiple sources and then combines it into a single output.

B. Placement of Intermediary Services

The next logical question is to identify ‘positions’ where to place intermediary services. As a general rule, intermediary services are needed wherever there is a link between actions of different services. In practice, *forwarding*, *replication* and *join* services could be offered by the same node, managed by the service integrator.

An output slot may be associated with more than one action at the same time or at different times. In this case, the slot value has to be replicated and synchronized with different actions. In the coordinated structure such points will appear as *submitOrder:*⟨*<SubmitOrderResp, (shipGoods1*∪ *shipGoods2)>*⟩ where the output is coordinated with several other actions. In this example, two warehouses are involved. Because we do not assume message templates to match, synchronization points are modeled using service intermediaries. One-to-One synchronization point is modeled by a *forwarder*, One-to-Many is modeled by a replicator. The typing of message templates/envelops is not fixed in that the intermediary services can transform between different message templates to satisfy service interfaces.

X. Conclusions And Future Work

Smooth coordination of services is faced with three interrelated concerns of *compatibility, adaptability* and *replaceability*. The key questions are if two services are *compatible* such that one can *replace* the other and if not what is required to *adapt* the services such that they work together. Our design approach considers a service-oriented system as a collection of activities where coordination of different services is aimed at performing a given activity with a set of possible outcomes.

To deal with adaptability, we explicitly separate service outputs from *s*emantic implication or *g*uarantees. Semantic implications provide means of coordinating activities over actions with several outputs. It answers the question, what does an action output imply in terms of the coordinated activity? In other words, outputs of service actions classified in terms of the expected result of the coordinated activity. The outputs of an activity provide classes over the outputs of actions involved in a particular activity.

We have provided a layout for a systematic methodology for constructing service-oriented systems. The approaches address some of the compatibility issues through its construction based on the notion of activities and guarantees that are used to form classes over action outputs. We plan to provide a more firm compatibility model that address compatibility at the level of services within the construction approach. We intend to automate the above process such that the task of service integrator is reduced to classification of action outputs in terms of coordinated activities, and minor adjustments on transformation of messages where inputs to do not exactly match with outputs.

References

[1] J. L. Bell, "Category theory and foundations for mathematics," *The British Journal for the Philosophy of Science*, vol. 32, no. 4, pp. 349–358, 1981.

[2] A. Asperti and G. Longo, *Categories, types, and structures: an introduction to category theory for the working computer scientist*. Cambridge, MA, USA: MIT Press, 1991.

[3] D. Berardi, D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Mecella, "Automatic composition of e-services that export their behavior," in *ICSOC*, 2003, pp. 43–58.

[4] R. Hamadi and B. Benatallah, "A petri net-based model for web service composition," in *ADC*, 2003, pp. 191–200.

[5] M. Mazzara and I. Lanese, "Towards a unifying theory for web services composition." in *WS-FM*, 2006, pp. 257–272.

[6] H. R. M. Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati, "Semi-automated adaptation of service interactions," in *WWW*, 2007, pp. 993–1002.

[7] D. Berardi, D. Calvanese, G. D. Giacomo, R. Hull, and M. Mecella, "Automatic composition of web services in colombo." in *SEBD*, 2005, pp. 8–15.

[8] J. Rao and X. Su, "A survey of automated web service composition methods," in *SWSWPC*, 2004, pp. 43–54.

[9] J. S. Zhongnan Shen, "On completeness of web service compositions," *icws*, vol. 0, pp. 800–807, 2007.

[10] W. Sanjiva and C. Francisco, "Business processes: Understanding bpel4ws," http://www.ibm.com/developerworks/library/ws-bpelcol1/, 2002, part 1. IBM developerWorks, Aug 2002.

[11] Gottschalk, K. Graham, S. Kreger, and J. H. Snell, "Introduction to web services architecture," *IBM systems Journal.*, vol. 2, no. 41, pp. 170–177, 2002.

[12] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer, "Simple object access protocol (soap)," http://w3c.org/TR/SOAP, June 2003, version 1.2.

[13] C. Ba, M. H. F. Alves, and M. A. Musicante, "Composing web services with pews: A trace-theoretical approach," in *ECOWS*, 2006, pp. 65–74.

[14] WS-I, "Supply chain management use case model," http://www.ws-i.org/SampleApplications/SupplyChainManagement/2003-12/SCMUseCases1.0.pdf, 2003.

[15] ——, "Sample application supply chain management architecture," http://www.ws-i.org/SampleApplications/SupplyChainManagement/2002-11/SCMArchitecture-0-11-WGD.pdf, 2002.

[16] D. Beyer, A. Chakrabarti, and T. A. Henzinger, "Web service interfaces," in *WWW*, 2005, pp. 148–159.

[17] D. Beyer, A. Chakrabarti, T. A. Henzinger, and S. A. Seshia, "An application of web-service interfaces," in *ICWS*, 2007, pp. 831–838.

[18] J. L. Fiadeiro, A. Lopes, and L. Bocchi, "A formal approach to service component architecture," in *WS-FM*, 2006, pp. 193–213.

[19] A. Brogi, C. Canal, E. Pimentel, and A. Vallecillo, "Formalizing web service choreographies," *Electr. Notes Theor. Comput. Sci.*, vol. 105, pp. 73–94, 2004.

[20] D. M. Yellin and R. E. Strom, "Protocol specifications and component adaptors," *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 2, pp. 292–333, 1997.

[21] J. L. Fiadeiro, "Algebraic support for service-oriented architecture," in *AMAST*, 2002, pp. 75–82.

[22] J. L. Fiadeiro and A. Lopes, "Algebraic semantics of coordination or what is in a signature?" *Lecture Notes in Computer Science*, vol. 1548, pp. 293–307, 1999.

[23] R. Hull, M. Benedikt, V. Christophides, and J. Su, "E-services: a look behind the curtain," in *PODS*, 2003, pp. 1–14.

[24] B. Benatallah, F. Casati, and F. Toumani, "Representing, analysing and managing web service protocols," *Data Knowl. Eng.*, vol. 58, no. 3, pp. 327–357, 2006.

[25] I. H. Krüger and R. Mathew, "Systematic development and exploration of service-oriented software architectures." in *WICSA*, 2004, pp. 177–187.

[26] J. A. Goguen, "A categorical manifesto." *Mathematical Structures in Computer Science*, vol. 1, no. 1, pp. 49–67, 1991.

[27] J. F. Costa1, A. Sernadas1, C. Sernadas1, and H. D. Ehrich, "Object interactions," in *Mathematical Foundations of Computer Science*. London, UK: Springer-Verlag, 1992, pp. 200–208.

[28] B. H. Liskov and J. M. Wing, "A behavioral notion of subtyping," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 6, pp. 1811–1841, 1994.

[29] OASIS, "Web services business process execution language version 2.0," 2007, http://www.w3.org/TR/wsdl20/, last accessed oct,2008.

[30] Z. Shen and J. Su, "Web service discovery based on behavior signatures." in *IEEE SCC*, 2005, pp. 279–286.

[31] J. L. Fiadeiro and T. Maibaum, "Categorical semantics of parallel program design," *Sci. Comput. Program.*, vol. 28, no. 2-3, pp. 111–138, 1997.

[32] B. Benatallah, F. Casati, D. Grigori, H. R. M. Nezhad, and F. Toumani, "Developing adapters for web services integration," in *CAiSE*, 2005, pp. 415–429.

View publication stats