Service-oriented Design: A Multi-viewpoint Approach

Remco Dijkman[1,2], Marlon Dumas[2]

[1] Centre for Telematics and Information Technology

University of Twente, The Netherlands

r.m.dijkman@utwente.nl

[2] Centre for Information Technology Innovation

Queensland University of Technology, Australia

m.dumas@qut.edu.au

Abstract

As the technology associated to the "Web Services" trend gains significant adoption, the need for a corresponding design approach becomes increasingly important. This paper introduces a foundational model for designing (composite) services. The innovation of this model lies in the identification of four interrelated viewpoints (interface behaviour, provider behaviour, choreography, and orchestration) and their formalization from a control-flow perspective in terms of Petri nets. By formally capturing the interrelationships between these viewpoints, the proposal enables the static verification of the consistency of composite services designed in a cooperative and incremental manner. A proof-of-concept simulation and verification tool has been developed to test the possibilities of the proposed model.

1. Introduction

With the increasing use of software applications for the daily conduct of business, the need to link software applications of business partners with minimal effort and in short timeframes is becoming ever more evident (Bussler 2003). Concomitant with the development of this need and greatly motivated by it, *Service-oriented Computing (SoC)* is emerging as a promising paradigm for enabling the flexible interconnection of autonomously developed and operated applications within and across organizational boundaries (Alonso et al. 2003).

SoC is a distributed application integration paradigm in which the functionality of existing applications (the *services* that they provide) is described in a way that facilitates its use in the development of applications which integrate this functionality. The resulting integrated applications can themselves be exposed as services, leading to networks of interacting services known as *service compositions or composite services* (Casati & Shan 2001, Benatallah et al. 2002). At present, SoC is mainly associated with its enabling technology (e.g. SOAP, WSDL, WS-Security, and BPEL4WS). This technology enables businesses to describe the services that they offer (generally in an XML-based form), to publish these descriptions online, to find other services based on their descriptions, and to build applications using these services. We argue that, as in other areas of computing, the technology for implementing and executing services should be complemented by modelling languages, methods, and techniques supporting their design. We call this set of languages, methods, and techniques *Service-oriented Design (SoD).* SoD is needed to aid in the communication between application architects as well as between application and enterprise architects. It is also needed to verify the conformance of services to their requirements and to enable a model-driven approach to service development and composition.

SoC brings along a number of specific requirements over previous paradigms (e.g. object and component-oriented) which unavoidably need to be taken into account by any SoD approach:

1. **High autonomy:** As services are expected to be developed by autonomous teams, SoD is an inherently collaborative process involving multiple stakeholders from different organizational units. This raises the issue that certain organizational units may opt not to reveal the internal business logic of their services to others, making it difficult (yet indispensable) to ensure global consistency.

2. **Coarse granularity:** Services are highly coarse-grained, at least more so than objects and components (Szyperski 2003). Often, a service maps directly to a business object or activity (e.g. a purchase order or a flight booking service). It follows that the design of services (and in particular composite ones) is a complex activity. It involves reconciling disparate aspects such as the involved providers and consumers, their interfaces, interactions, and collaboration agreements, their internal business processes, data, and (legacy) applications.

3. **Process awareness:** As services often correspond to business functionality exported by an organizational unit, they are likely to be part of long-running interactions driven by explicit process models (Aalst 2003). Hence, SoD should take into account the business processes as part of which services operate and interact, and in particular, the integration (or retrofitting) of services into business processes. This effectively places SoD at the crossroads between software and enterprise design.

In light of these requirements, we argue that before constructing an approach for SoD or adapting an existing approach from another area (e.g. defining a UML profile) we need to develop an understanding of the fundamental *concepts* of SoC. We call such a set of concepts and their interrelationships a *service-oriented design model.* The goal of this paper is to motivate and propose a SoD model that takes into account the above requirements and satisfies the following general principles:

1. **Relevance:** They should cover the properties that developers of services and compositions of services find relevant.

2. **Automated support:** They should provide a basis for reasoning about (composite) services, e.g. for simulation and verification purposes.

3. **Technology-independence:** They should abstract from specific implementation technologies (e.g. WSDL, SOAP and BPEL4WS) and specific platforms.

To cope with these requirements, the proposed design model adopts a *multi-viewpoint approach*, along the lines of RM-ODP (ITU-T/ISO 1994–1997) and IEEE 1471 (IEEE Architecture Working Group 2000). By supporting multiple modelling viewpoints, it is possible to break down a service design into smaller more manageable parts which are handled by different stakeholders (thereby addressing the "autonomy" and "granularity" requirements). In accordance with the "relevance" requirement, the supported viewpoints correspond to different aspects of SoC covered within ongoing standardization and development efforts, namely *service interfaces*, *providers*, *choreographies*, and *orchestrations*. Importantly, the proposed design model recognizes the need to maintain consistency across viewpoints. To this end, all viewpoints and their interrelationships are defined in a common process modelling formalism (thereby addressing the "process awareness" requirement). Using a formalism for process modelling allows us to abstract from specific technologies (technology-independence) and provides a foundation for verifying the resulting service designs (automated support). Specifically, we chose Petri nets as a process modelling formalism, while acknowledging that there are other alternatives, such as CSP

(Milner 1989), LOTOS (van Eijk 1989), and pi-calculus (Milner 1999), we chose to use Petri nets.

Also, while acknowledging that an SoD model should be suitable for describing all aspects relevant to the development of services, in this paper we focus on the *control-flow aspect*. Informally, this aspect covers the order in which service interactions and internal tasks occur. We limit ourselves to this aspect because it is one of the most controversial (see e.g. the discussions on the choreography vs. orchestration dichotomy (Pelz 2003)) and because it constitutes a foundation upon which other aspects (e.g. the *data* and *transactional* aspects) can be layered.

The paper is structured as follows. Section 2 lays the formal background for the proposal in the form of a "core model". Section 3 introduces the four viewpoints recognized by the design model and defines them with reference to the core model. Section 4 presents and formalizes a number of relations between viewpoints and sketches a method for ensuring the consistency of composite service designs using these relations. Finally Section 5 compares the proposal to related work and Section 6 presents our conclusions.

2. Core Model for Service-oriented Design

This section sets up a (formal) basis for Service-oriented Design. It presents this basis in the form of a meta-model that defines the *terms* that are used in Service-oriented Computing and Design, the *meaning* of these terms and the *relations* that these terms have with each other. The meaning of the terms is refined by a Petri-net based model that defines the dynamic aspects of services and compositions of services from a control-flow perspective. We derived the meta-model by generalization from the concepts that are supported by existing and proposed (web) service description standards, such as WSDL (W3C 2003), WSCI (Arkin et al. 2002b), BPEL4WS (BEA Systems et al. 2003), BPML (Arkin et al. 2002a) and BPSS (UN/CEFACT & OASIS 2001). This section first explains the static aspects of service-oriented design with a meta-model. It then explains the dynamic aspects of service-oriented design with a Petri net based addition to this meta-model. Finally, it presents an example in which the meta-model is used.

2.1 Static Aspects

Existing service description languages make their services available by specifying the interactions in which they can engage with their environment. The interactions that they specify correspond to communication mechanisms that are supported by the service platforms for which they are intended. Examples of these communication mechanisms are one-way message passing, remote procedure call and transaction. All of the description languages that we investigated define the communication mechanisms that they use in terms of the sending and receiving of messages. The one-way message passing mechanism is defined as one partner sending a message and another partner receiving that message. The remote procedure call mechanism is defined as one partner sending a request message, another partner receiving that message and optionally sending a response message or one of a set of fault messages and the first partner receiving that response or fault. In accordance with what the description languages do, we use one-way message passing as our basic mechanism for communication. We call sending a message a *send event*, receiving a message a *receive event* respectively and a combination of a send event by one partner and a receive event by another partner an *elementary interaction* (as opposed to a *complex interaction* that may also represent a remote

procedure call or transaction). Since, we only deal with message passing from this point on, we simply refer to an elementary interaction as *interaction*.

The different description languages define their behaviour in terms of (flow) relations between tasks. A task may correspond to a particular pattern of send and receive events. We call such tasks *communication tasks* and the pattern of send and receive events to which they correspond a *messaging pattern*. Alternatively, tasks may represent an internal activity performed by a single business partner. We call such tasks *internal tasks*. The (flow) relations between the interactions in which service providers can engage are implicitly defined by the flow relations between the different tasks. All description languages predefine particular types of communication tasks that predefine particular messaging patterns. For example, BPEL4WS defines the 'invocation' task type that defines the message pattern in which a send event occurs first, after which a choice of receive events is expected, where one receive event corresponds to a normal message reception and the other receive events correspond to the reception of a fault message (obviously, this defines the messaging pattern that the caller of a remote procedure call displays, hence the name invocation task).

Different description languages vary with respect to the communication and messaging patterns that they support and a large variety of communication patterns has been documented, for example by Hohpe & Woolf (2004). It is not our goal here to identify all communication task types and the patterns to which they correspond, but rather to provide a semantic basis for understanding and analyzing service oriented designs. Therefore, we only use communication tasks for grouping messaging events and we define the behaviour of a business partner by defining the (flow) relations between the interactions rather than between the communication tasks. The benefit of this approach is that it can be used to analyze designs from all description languages. Moreover, it is possible to define mappings from the existing description languages to our design model by 'exploding' the tasks that the design model supports into the interactions that they represent. The drawback of this approach is that behaviours that are modelled with it can not be mapped directly onto a description language. Rather, interactions must first be grouped into tasks, such that the messaging patterns encapsulated by the tasks match the messaging patterns supported by the description language. Also, the control flow between the tasks must match a pattern supported by the target description language. The analyses of the control-flow and communication patterns supported by BPEL4WS, BPML and WSCI (Wohed et al. 2003) can help with this.

The final term that we have to define, before presenting our meta-model, is that of *role*. A role is a prescribing behaviour that can be performed any number of times by any service provider, concurrently or successively. Every time a service provider fulfils a role, we create an *instance* of that role. Hence, multiple instances of the same role can exist at one moment in time and each of these instances represents a service provider that fulfils that role. As an example, consider the role of a mortgage broker in a collaboration. Any service provider may fulfil that role, including a service provider that also fulfils the role of 'bank' in the same collaboration. If a service provider fulfils the role of the mortgage broker, it has to act as one (i.e. observe the behaviour that the role prescribes). Hence, any number of mortgage brokers can exist simultaneously and each of them is represented by a role instance.

These observations lead to the following meta-model which captures the static aspects of the proposed SoD model.
SE is the set of all send events.
RE is the set of all receive events.
E is the set of all events such that $SE \subseteq E$, $RE \subseteq E$, $SE \cap RE = \varnothing$ and $SE \cup RE = E$.
$I \subseteq SE \times RE$ is the set of (elementary) interactions.
R is the set of roles.
RI is the set of role instances.
IT is the set of tasks that are only available internally to one role or another.

CT is the set of tasks that represent messaging patterns for interaction with other roles.
event: $E \rightarrow R$, where *event*(*e*) = *r* represents that role *r* engages in messaging event *e*.
communication: $CT \rightarrow R$, where *communication*(*ct*) = *r* represents that communication task *ct* is of role *r*.
internal: $IT \rightarrow R$, where *internal*(*it*) = *r* represents that internal task *it* is of role *r*.
pattern: $E \text{ -/-> } CT$, where *pattern*(*e*) = *ct* represents event *e* is a part of the communication pattern represented by communication task *ct*.
instance: $RI \rightarrow R$, where *instance*(*ri*) = *r* represents that role instance *ri* is an instance of role *r*.

The following constraints must hold on the meta-model:
(1) $\forall se, re_1, re_2 \in E: (se, re_1) \in I \wedge (se, re_2) \in I \Rightarrow re_1 = re_2$ and
(2) $\forall se_1, se_2, re \in E: (se_1, re) \in I \wedge (se_2, re) \in I \Rightarrow se_1 = se_2$, representing that each send event and receive event may be used in an interaction only once, therewith prohibiting broadcast and multicast. As we did not find broadcast or multicast in any of the investigated languages (Wohed et al. 2003), we consider this a reasonable restriction.
(3) $\forall e \in E, r \in R, ct \in CT: \ event(e) = r \wedge pattern(e) = ct \Rightarrow communication(ct) = r$, representing that if a role engages in an event that is part of a communication pattern then that communication pattern must be part of the same role.

For illustration purposes, the diagram from figure 1 gives a graphical representation of the above meta-model. Each class in the diagram corresponds to a set in the formal model and each relation in the diagram corresponds to a relation in the formal model. The partition of the set of events into send events and receive events is represented by an inheritance relation and the construction of an interaction from a send and a receive event is represented by two aggregation relations.

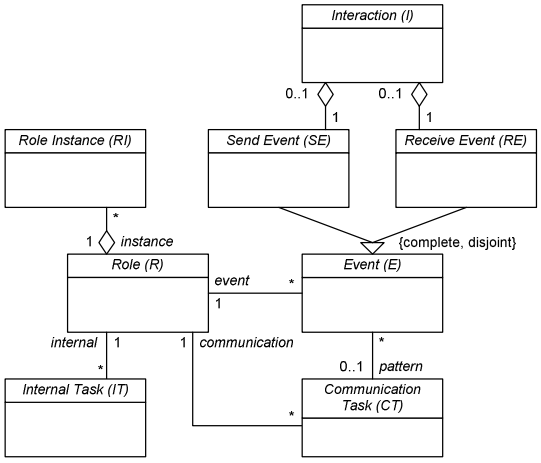

Figure 1. A Graphical Representation of the Meta-Model

To construct a particular model, the sets and relations in the meta-model are populated and we can refer to the classes and relations as if they were sets (e.g. we refer to the roles of a particular model as *R)*. To avoid confusion between the sets and relations of different models we sometimes subscript them with the model name (e.g. R_M designates the roles of a model ***M***).

2.2 Dynamic Aspects

The above meta-model is complemented by a marked labelled Petri net *N* capturing the dynamic aspects of the proposed SoD model. Specifically, a marked labelled Petri net that describes the dynamic aspects of a service-oriented design is a five-tuple:

(P, T, F, L, M), such that:
P is the set of places in the net,
T is the set of transitions in the net, where $P \cap T = \varnothing$,
$F \subseteq (P \times T) \cup (T \times P)$ is the flow relation between places and transitions,
L: $T \rightarrow E \cup IT \cup \{\tau\}$ is the labelling function that associates each transition with an event, an internal task or the 'silent' label τ that represents that nothing of interest (or at least no internal task or event) is happening,
M: $P \rightarrow$ Nat is the marking of the net. M^0 is the initial marking M that represents the initial situation in a certain behaviour.

Since the Petri net is defined to be a part of the meta-model, we may refer to the Petri net of a particular model $\boldsymbol{M}$ in the same way as we refer to, for example, the roles of that model. Hence we may refer to it as: $N_{\boldsymbol{M}}$. We can also refer to the properties of the Petri net that defines the behaviour of a model, by directly subscripting that with the name of the model. For example, the places of the Petri net that belongs to model $\boldsymbol{M}$ are $P_{\boldsymbol{M}}$. We define *Nets* as the set of all possible marked labelled Petri nets.

To relate the Petri net to the service-oriented design of which it is a part, the labels of the Petri net are taken from the set of events and internal tasks of the meta- model defined above. The set of possible labels does not include the communication tasks of the meta- model, since the behaviour can not be defined on them for reasons explained above. To further relate a service oriented design to the Petri net that describes its behaviour, we define that an interaction from the model corresponds to a place in the Petri net. This place has a flow relation to it from the transition that represents the send event of the interaction and this place has a flow relation from it to the transition that represents the receive event of the interaction. Figure 2 graphically represents an interaction and the send event and receive event that this interaction incorporates.

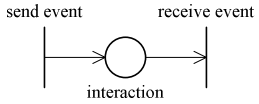

Figure 2. A Petri Net that Represents an Interaction

Therefore, formally, we say that the set of Petri net places in a model must include the set of all interactions in that model:
(4) $I \subseteq P$
Also, we say that if a place represents an interaction, then it may only have one transition in its pre-set and one transition in its post-set and this transitions must be labelled with the send event and the receive event of the interaction, respectively:
(5) if there exists an interaction $(se, re) \in I$ then there exist unique transitions t_1 and t_2, such that $\bullet(se, re) = \{t_1\} \wedge (se, re)\bullet = \{t_2\} \wedge L(t_1) = se \wedge L(t_2) = re$

On the basis of this, we define the *subnet* function (*subnet*: $R \rightarrow \wp(P \cup T)$). This function relates a role to the set of places and transitions that define its behaviour. All transitions must be part of a subnet and a place may only not be part of a subnet if it represents an interaction. Formally, for all roles r, places p and transitions t:
(6) if $(L(t) \in E \wedge event(L(t)) = r) \vee (L(t) \in IT \wedge internal(L(t)) = r)$ then $t \in subnet(r)$, representing that a transition t is in the subnet of role r if the label (an event or an internal task) of t is of r, as defined in the static part of the meta-model.
(7) if $\exists p' \in subnet(r)$: $p' \in (\bullet t \cup t\bullet)$ then $t \in subnet(r)$ representing that a transition t is in the subnet of role r if there is a place in its pre-set or post-set that is in the subnet of r.
(8) if $p \notin I \wedge \exists t' \in subnet(r)$: $t' \in (\bullet p \cup p\bullet)$ then $p \in subnet(r)$ representing that a place p is in the subnet of role r if p does not represent an interaction and there is a transition in its pre-set

or post-set that is in the subnet of r.
Algorithmically, the subnet of a role r can be constructed by repeatedly adding places and transitions that satisfy the above constraints, until a stable situation is reached.

Finally, each role r is associated with an initial marking M^0_r. The initial marking of a role can only be defined on places that are in the subnet of the role: M^0_r: $(P \cap subnet(r)) \rightarrow Nat$. Each time an instance of role r is created, a set of tokens that is identified by the initial marking of that role is placed on the Petri net. Therefore, the initial marking of the Petri net of a service-oriented design equals the union of the initial markings of the roles of that design, where each initial marking is multiplied by the number of instances of the corresponding role. Formally: (9) for all places p and roles r: if $p \in subnet(r)$ then $M^0(p) = M^0_r \cdot |\ instances(r)\ |$, representing that if a place p is in the subnet of a role r then its initial marking is equal to the initial marking that r assigns to it, times the number of instances of r. This constraint works with the assumption that each place is associated with at most one role.

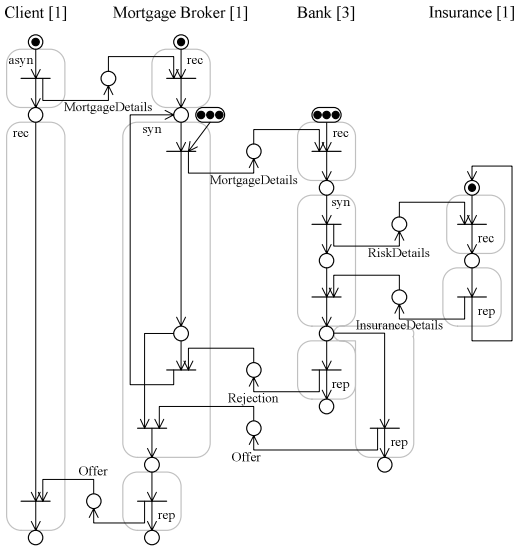

Figure 3. An Example of a Service-oriented Design

2.3 Example

Figure 3 illustrates our approach. It shows a collaboration between four service providers to sell a mortgage to a client (we consider a client a service provider that only uses services of other service providers). The figure unifies the static and the dynamic aspects of the design in one drawing. It clearly shows the Petri net that represents the dynamic aspects of the design. It shows the static aspects of the design as annotations of this Petri net. The roles involved in the design are represented as four labels above the model: client, mortgage broker, bank and insurance. Each role is suffixed with a number between square brackets that indicates the number of instances of that role. Hence, only the bank role has more than one instance. Verify that, because there are three instances of the bank role, the initial marking associated with the bank role has only one token on the topmost place. In the model there are three tokens on that place, because there are three instances of the bank role. An event that is associated with a role is represented as a transition below that role's name. Events are grouped into communication tasks by rounded rectangles. The labels of these rounded rectangles represent a particular type of communication task, as we will explain later on in this section. An interaction is represented as a place that is not below any role name. For reasons of simplicity, we have not labelled the send and receive events in the example. Instead, we have annotated each interaction with the message that is exchanged and we say that the send event that

corresponds to the interaction is labelled with ‘send<message>’ and the receive event with ‘receive<message>’. Also, for reasons of simplicity, we have not modelled that the client, the mortgage broker and the bank (may) repeat themselves after they have performed their tasks. The model shows that a client sends a message to the mortgage broker. This message contains details about the mortgage that the client wants. The mortgage broker forwards the mortgage details to a bank. The bank then contacts the insurance company to see if it can get a suitable insurance for the mortgage and the insurance company replies with a proposal. Based on this proposal, the bank sends a mortgage offer or a rejection back to the mortgage broker. If the mortgage broker receives a rejection, it will retry the procedure by forwarding the mortgage details to another bank. If it receives an offer, it will forward the offer to the client. The mortgage broker can repeat this procedure at most twice (because of the place with three tokens that is associated with it).

Because our approach focuses on the control flow aspects of a service oriented design, we do not express constraints related to data exchanged or constraints related to the identity of role instances. An example of this latter type of constraint is that the mortgage broker contacts a *different* bank each time that it requests an offer for a particular client. If one wants to be able to express constraints on data or identity, one should use coloured Petri nets instead of basic Petri nets. One can then use the analysis techniques and tool support associated with coloured Petri nets in addition to the analysis techniques for verification of control flow from different viewpoints that we propose below. In coloured Petri nets a place has a data type and tokens on that place carry a value of the type associated with that place. Transitions may put constraints on the value that tokens may have for it to fire and may perform an operation on the values of some tokens to put the result of that operation on a target place in the form of a new token. For illustration purposes only, figure 4 shows a part of the design from figure 3 that has been elaborated as a coloured Petri net. In the figure, initially, the transition ‘sendMortgageDetails’ can fire. When it fires, it takes a token from the place that has the data type ‘MortgageDetails’ and assigns its value to the variable ‘m’. In the case shown in the figure, this can only be the token with value ‘m1’. Similarly, it takes a token from the place that has the data type ‘Bank’ and assigns its value to the variable ‘b’. Let’s say it takes the token with value ‘b1’. It then puts a token on the place that has the data type ‘Message’ with the value that is formed by combining the values associated with ‘b’ and ‘m’ into a tuple. Therefore in our case, it puts a token with value ‘(b1,m1)’ on the place. The transition ‘recvMortgageDetails’ can only fire if a token from the place that has the data type ‘Bank’ can be found that carries the same value as the first component of the tuple that is on the place that has the data type ‘Message’. This represents a constraint on the identity of the instance of the bank role. Similarly, the constraint on the ‘sendRejection’ transition represents a data constraint on the mortgage details that the bank role receives. Intuitively, it specifies that a bank sends a rejection if the mortgage details that it received are not acceptable.

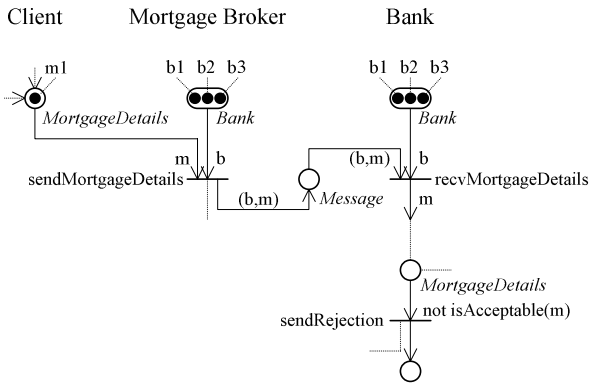

Figure 4. An Example of a Service Oriented Design with Data and Identity

In figure 3, we grouped the send and receive events into communication tasks, by drawing rounded rectangles around the send and receive events. In the example, we made each communication task correspond to a messaging pattern that is pre-defined in BPEL4WS. Figure 5 shows the messaging patterns from BPEL4WS and the task types to which they correspond. The task type of a particular communication task is indicated in figure 3 with an abbreviation. We claim that the example from figure 3 can be translated into a BPEL4WS description, in which each communication task from the figure corresponds to a task in the BPEL4WS description.

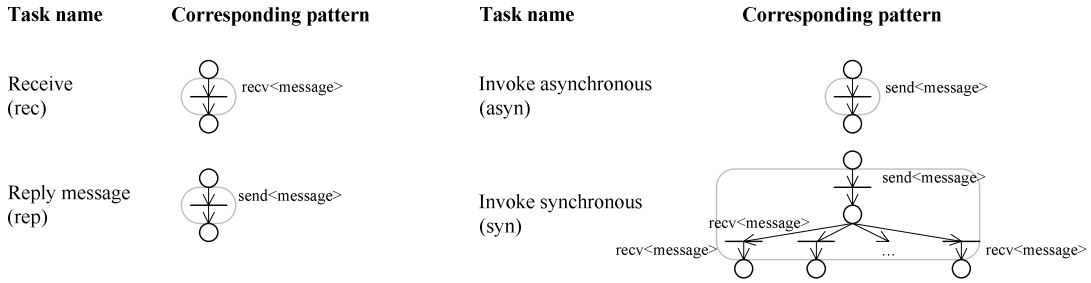

Figure 5. The Tasks from BPEL4WS and the Messaging Patterns that they Represent

3. Viewpoints in Service-oriented Design

From existing approaches and techniques for the development of web services we have derived that there are a number of viewpoints from which we can describe the control-flow aspect of web services. Specifically, the viewpoints that we identified are: the choreography viewpoint, the interface behaviour viewpoint, the provider behaviour viewpoint and the orchestration viewpoint. We further explain each viewpoint below by giving a definition for it, by explaining the goal that a designer has to construct a design from it and by showing its relation to the core model of the previous section.

3.1 Choreography

A *choreography* is a collaboration between some enterprise service providers and their users to achieve a certain goal. It describes the tasks that enterprise service providers perform in order to achieve that goal and the interactions between enterprise service providers and providers and users that are the result of the execution of these tasks. A choreography also describes when tasks and interactions may happen, by describing the (flow) relations that exist between tasks, between interactions and between tasks and interactions. Since a choreography is a collaboration, it only describes tasks that involve communication between the parties involved. It does not describe (sub-)tasks that service providers perform internally to realize a service that they perform for others, because these tasks are not essential to the collaboration (a business partner is not interested in *how* the other realizes its service, only in *that* it performs its service). Also, since a choreography is goal-focused, it does not focus on a particular service provider, by only describing the interactions and tasks in which that service provider can participate. Rather, it describes all interactions and tasks between service providers and between service providers and service users that contribute to the common goal. Figure 3 is an example of a choreography. It can be seen that the choreography from this figure can not be described if we focus on a single service provider.

A choreography covers the perspective of a stakeholder that wants to have an overview of a collaboration. A goal for constructing such an overview is to be able to verify whether the

joint behaviour of some service providers or the behaviour of a particular service provider conforms to the originally intended collaboration. Another goal for constructing such an overview is to have a standard collaboration in which individual service providers can indicate the interactions that they can participate in and the tasks that they can perform. The choreography then provides a starting point from which a concrete collaboration can be set up.

From these observations, we derive that a design from the choreography viewpoint must respect the following constraint on the generic design model from the previous section:
(10) $IT = \varnothing$, representing that internal tasks are not considered.

3.2 Interface Behaviour

An *interface behaviour* is the behaviour of a particular service provider or service user in its communication with a single other service provider or service user to achieve a particular goal. Since an interface behaviour only describes the behaviour of a single service provider, it consists of only one role instance. Since it is related exclusively to one role instance, it also deals with only one role, because a role instance is an instance of exactly one role. An interface behaviour only describes send and receive events. It does not describe interactions, because these exist between different roles. Since an interface behaviour only describes behaviour that is related to communication with other parties, it does not describe internal tasks.

Often, a distinction between *provided* and *required* interface behaviours is made. The distinction between these different interfaces is based on the types of communication tasks that an interface is allowed to use and the way in which these communication tasks can be related. A provided interface behaviour can only use communication task types in such a way that a receive event always occurs first, after which an optional choice of send events may occur. The idea is that the receive event corresponds to the *request* for a certain business function to be performed and the send events correspond to the possible *responses* that may be the result of performing this business function. In terms of the task types supported by BPEL4WS that are shown in figure 5, a provided interface behaviour must be composed of receive and reply tasks, where the replies are always coupled to a receive (BPEL4WS calls this correlation of replies to receives). A provided interface behaviour is often called a *service*. A required interface behaviour can only use communication task types in such a way that a send event always occurs first, after which an optional choice of receive events may occur. The idea is that a required interface works as the counterpart of the provided interface and that therefore the send event corresponds to the *request* for a certain business function to be performed and the receive events correspond to the possible *responses* that may be the result of performing this business function. In terms of the task types supported by BPEL4WS that are shown in figure 5, a required interface behaviour must be composed of asynchronous or synchronous invoke tasks.

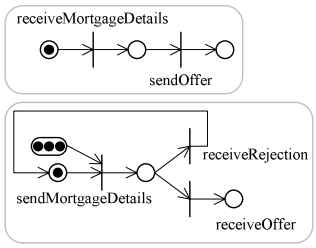

Figure 6. Two Examples of Interface Behaviour.

Figure 6 shows two examples of interface behaviour. The interface behaviours have been identified by drawing a rounded rectangle around them. The first example shows the interface behaviour of a mortgage broker as it communicates with its clients and the second example shows the interface behaviour of the mortgage broker as it communicates with banks.

An interface behaviour covers the perspective of the service provider that provides a web service or of the service provider that wants to use a web service that is provided by another service provider. The goal of specifying an interface behaviour is to make clear what an enterprise service provider can do for a particular business partner or expects of a particular business partner.

From these observations, we derive that a design from the interface behaviour viewpoint must respect constraint (10) specified in the choreography viewpoint, as well as the following constraints:
(11) $|RI| = 1$, representing that an interface behaviour deals with only one role instance (and therefore with only one role).
(12) $I = \varnothing$, representing that interactions are not considered in the interface behaviour viewpoint, but only send and receive events.

3.3 Provider Behaviour

A *provider behaviour* is the behaviour of a particular enterprise service provider or enterprise service user in its communication with all its business partners. The provider behaviour of an enterprise service provider is similar to the interface behaviours of a service provider. However, it is not limited to the collaboration with a single business partner, but rather describes the collaborations with all business partners. The same description techniques that apply to interface behaviour description therefore also apply to provider behaviour description. Hence, a provider behaviour corresponds to the events and communication tasks that are specified in the interface behaviours of a particular service provider, the relations between the events and tasks within individual interface behaviours and the relations between events and tasks from different interface behaviours.

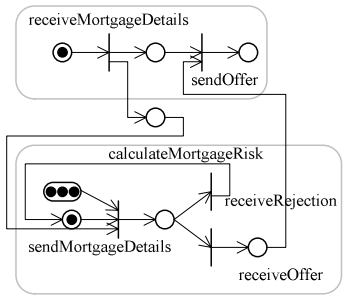

Figure 7. An Example of a Provider Behaviour.

Figure 7 shows an example of a provider behaviour. The example shows the provider behaviour of a mortgage broker. The example clearly shows that the provider behaviour of the mortgage broker was derived from the interface behaviours of the mortgage broker from figure 6 by adding a place and three flow relations. Adding places and flow relations to a design adds constraints on when events can happen. Hence we refer to this method of constructing provider behaviour as *constraint oriented structuring* (Quartel et al. 1997). The benefit of constraint oriented structuring is that it clearly separates the relations between events within a single interface from the relations between events in separate interfaces. In that way, the original interface behaviours remain in tact and clearly recognizable. We will further elaborate on this in the next section.

A provider behaviour description covers the perspective of the enterprise service provider that fulfils that provider behaviour. The reason for constructing a service description can be, for example, for the enterprise service provider to know exactly what it has to implement. This information is useful when the service provider wants to change the implementation, while keeping the implemented provider behaviour intact, or when the provider behaviour was derived from a standard choreography that, by definition, does not prescribe how the service provider has to implement the provider behaviour.

The provider behaviour viewpoint has the same conceptual model as the interface behaviour viewpoint. Therefore, constraints (10) through to (12) from the choreography and the interface behaviour viewpoints apply. From this, we derive that the difference between interface behaviour and provider behaviour is purely a methodological one.

3.4 Orchestration

An *orchestration* is the behaviour that a service provider performs internally to realize a service that it provides. For this purpose, the service provider may communicate with other business partners at its required interfaces. In addition to the messaging events and the communication tasks that model the collaborations with other parties, an orchestration describes tasks that a service provider performs internally. Current description languages for orchestration limit the internal actions to data transformations, such as string and integer operations and type conversions. The reason for limiting the allowed internal actions is that orchestrations are intended to be executed by an orchestration engine and that, therefore, the internal tasks must be supported by such an engine. Because an orchestration is intended to be executed by an orchestration engine, it is also called an executable process.

Figure 8 shows an example of an orchestration. The orchestration introduces an internal task to the provider behaviour of the mortgage broker from figure 7. In the internal task, the risk of a mortgage is calculated, based on the details that are provided about the mortgage.

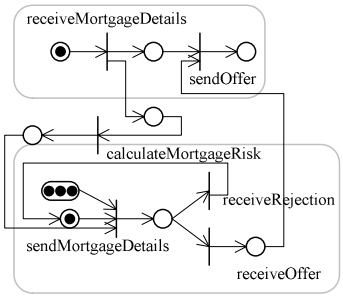

Figure 8. An Example of an orchestration.

An orchestration covers the perspective of the stakeholders that implement a provider behaviour in an orchestration engine. The goal for constructing an orchestration is to realize a web service.

The orchestration viewpoint has the same conceptual model as a the interface behaviour and provider behaviour viewpoint, except that constraint (10), which states that there are no internal tasks, does not apply.

4. Relations between the Viewpoints

In this section we discuss the relationships between the viewpoints that are identified in the previous section, both informally and formally. Subsection 4.1 gives an informal overview of the relations between the viewpoints. The subsequent subsections describe the relations between each pair of viewpoints more precisely in terms of relational operators and the properties that these operators exhibit.

4.1 Overview of the Relations between Viewpoints

Figure 9 shows an example that illustrates the relations between the different viewpoints.

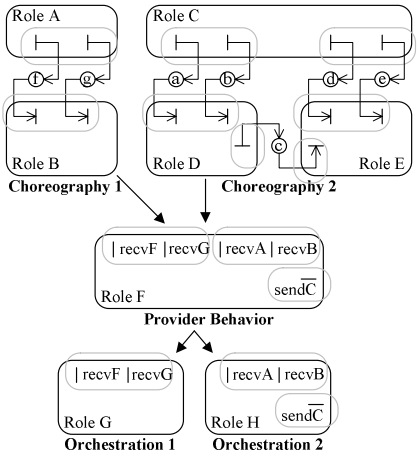

Figure 9. Relations between Viewpoints

The figure shows two choreographies. The choreographies show the interactions that occur between participating roles (and role instances). The choreographies imply the existence of certain interfaces. These interfaces are represented by rounded rectangles. Each interface groups the send and/or receive events that relate to the interactions of a role with a single other role.

The figure shows a single provider behaviour that engages in role B of choreography 1 and role D of choreography 2. Which roles a provider behaviour engages in is a design choice. Because of the chosen relation to the choreographies, the provider behaviour from the example includes the events of roles B and D. The provider behaviour can be partitioned into a number of interfaces. Although this partition is a design decision, the partition shown in the figure is the most logical choice, because this is the partition that is implied by the choreographies from which the provider behaviour was created.

The provider behaviour from the figure is realized by two orchestrations. We did not include internal tasks in the orchestrations, but normally orchestrations would include internal tasks. The orchestrations of the service provider partition the events from the provider behaviour. Although the partition into two orchestrations corresponds to the two choreographies in which the service provider engages, this is merely a design choice. One could easily envision a single orchestration that realizes the provider behaviour of the service provider.

In this section, we limit ourselves to these viewpoint relations. However, other relations can easily be envisioned (e.g. between an interface and a choreography). We claim that these relations can be expressed as transitive relations with the operators that are defined below.

4.2 Relating Provider behaviours to Choreographies

A choreography provides a point of verification, describing interaction patterns that service providers must conform to if they want to do business. A choreography can be used as a point of verification in two ways. We can use a choreography as a starting point and verify whether the behaviour of a particular service provider (a particular provider behaviour) conforms to the behaviour prescribed by the choreography. Also, we can derive the joint behaviour, or choreography, of a number of service providers (a set of provider behaviours) and verify that it makes sense. We do not describe below what we mean by a choreography that 'makes sense', but we do show how a choreography can be derived from a set of provider behaviours. This derived choreography can then be simulated and analyzed for deadlocks and other properties, depending on the designer's wishes, to verify that it makes sense.

4.2.1 Verify the Behaviour of a Single Service Provider

A single service provider can play a number of roles in a choreography. Hence, to verify the behaviour of a particular service provider, we focus on the behaviour of the roles that the service provider plays in a choreography. Moreover, we focus on events of interactions that these roles engage in with other roles. We call these events the events that are at the *service boundary* of the roles that the service provider plays. We focus on the events that are at the service boundary, because events that are inside the service boundary are invisible to the external observer (i.e. other roles). Therefore, the behaviour of a service provider conforms to the behaviour prescribed by a choreography, if it *at least* displays the behaviour at the service boundary. We can use the following approach to determine the behaviour at the service boundary of a set of roles *rs* in the context of a choreography *C*.

First, we determine the behaviour of the set of roles $rs \subseteq R_C$ that the service provider plays in the choreography. To do this we use the restriction operator. The restriction operator, $\backslash : Nets \times \wp T \rightarrow Nets$ delivers for a Petri net $\boldsymbol{P}$ and a set of transitions X, the Petri net that consists of these transitions, the places that have a pre-set and a post-set that is completely constituted of these transitions and the flow relations, the labels and markings that are defined on these places and transitions. Formally: $\boldsymbol{P} \setminus X$ is the Petri net for which:
$P_{\boldsymbol{P}\setminus X} = \{p \mid \bullet p \subseteq X \wedge p\bullet \subseteq X\}$
$T_{\boldsymbol{P}\setminus X} = T \cap X$
$F_{\boldsymbol{P}\setminus X} = \{(x, y) \mid (x, y) \in F \wedge x \in (P_{\boldsymbol{P}\setminus X} \cup T_{\boldsymbol{P}\setminus X}) \wedge y \in (P_{\boldsymbol{P}\setminus X} \cup T_{\boldsymbol{P}\setminus X})\}$
$L_{\boldsymbol{P}\setminus X} = \{(t, l) \mid (t, l) \in L \wedge t \in T_{\boldsymbol{P}\setminus X}\}$
$M_{\boldsymbol{P}\setminus X} = \{(p, m) \mid (p, m) \in M \wedge p \in P_{\boldsymbol{P}\setminus X}\}$
For a model of a choreography *C*, this operator can be used to restrict the behaviour N_C of that choreography to the behaviour of a particular role $r \in R_C$, using the formula:
$N_C \setminus (subnet(r) \cap T_C)$, or alternatively to the behaviour of a set of roles $rs \subseteq R_C$, using the formula: $N_C \setminus (\cup(map\ subnet\ rs) \cap T_C)$. This formula uses the *map* operator that applies a function to all elements of a set and delivers the resulting set. We define the function
$map: (X \rightarrow Y) \times \wp X \rightarrow \wp Y$, such that:
$map\ fn\ xs = \{fn(x) \mid x \in xs\}$

As an example, consider the choreography *C* from figure 3 and suppose that we want to restrict the choreography to the bank and insurance company roles {*Bank*, *Insurance*}. The transitions that the bank engages in are: $subnet(Bank) \cap T_C$. If we identify these transitions by their labels, we get the set of transitions: {*recvMortgageDetails, sendRiskDetails, recvInsuranceDetails, sendRejection, sendOffer*}. Similarly, the transitions that the insurance company engages in, identified by their labels, are: {*recvRiskDetails*, *sendInsuranceDetails*}. If we restrict the Petri net N_C of the choreography to the Petri net that contains only these

transitions $N_C \setminus \{$*recvMortgageDetails, sendRiskDetails, recvInsuranceDetails, sendRejection, sendOffer*, *recvRiskDetails*, *sendInsuranceDetails*$\}$, we get the diagram from figure 10.

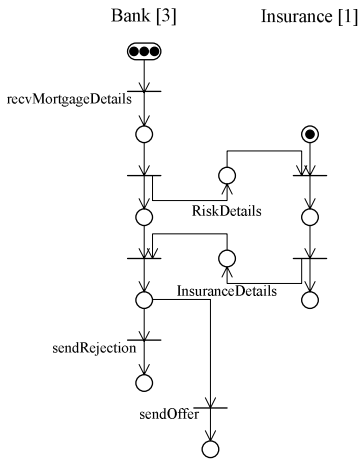

Figure 10. A Choreography Restricted to the Bank and Insurance Roles

A choreography can contain multiple instances of a particular role and the formula above returns all instances of that role. However, because we only want to compare the behaviour of a single service provider, we only want one instance of each role. We can obtain a single instance of a role r, by setting the marking of that role its initial marking: M^0_r. For this, we use the operator *mark*: $\boldsymbol{Ms} \times Nets \rightarrow Nets$, which delivers for a Petri net $\boldsymbol{P}$ and a marking M the Petri net $\boldsymbol{Q}$ in which the marking of $\boldsymbol{P}$ has been replaced by M (where $\boldsymbol{Ms}$ represents all possible markings). Formally:
mark M $\boldsymbol{P} = (P_{\boldsymbol{P}}, T_{\boldsymbol{P}}, F_{\boldsymbol{P}}, L_{\boldsymbol{P}}, M)$.
For a role r, the behaviour of a single instance of that role can then be obtained by the formula:
mark M^0_r $(N_C \setminus (subnet(r) \cap T_C))$. For a set of roles *rs*, the behaviour of a single instance of each of these roles can be obtained by the formula:
mark $(\cup_{r \in rs} M^0_r)$ $(N_C \setminus (\cup(map\ subnet\ rs) \cap T_C))$, where $\cup_{r \in rs} M^0_r$ delivers the marking that is the union of all initial markings of the roles from *rs*.

As an example, consider the (restricted) choreography from figure 10 in which there are three instances of the role bank and one of the role insurance. To derive the model in which there is only one instance of each role, we have to replace the marking of the net by the initial markings of the roles. The initial marking of the role bank only had a single token on the topmost place and the initial marking of the role insurance is the same as the marking that is shown. Hence, the marking from figure 11 is the marking in which there is one instance of the bank role and the insurance role.

Second, we determine the events inside the service boundary of the set of roles $rs \subseteq R_C$ that the service provider plays. To do this, we use the *IN* operator. The operator $IN_{\boldsymbol{M}}$: $\wp R_{\boldsymbol{M}} \rightarrow \wp E_{\boldsymbol{M}}$, delivers for a given set of roles *rs* all events that are inside the service boundary of the specified roles in the context of model $\boldsymbol{M}$. These are the events that relate to interactions these roles perform jointly:
$IN_{\boldsymbol{M}}(rs) = \cup\{\{se, re\} \mid (se, re) \in I_{\boldsymbol{M}} \wedge \exists\, r_1, r_2 \in rs\colon (se, r_1) \in event_{\boldsymbol{M}} \wedge (re, r_2) \in event_{\boldsymbol{M}}\}$

Finally, we abstract from the events inside the service boundary of the roles $rs \subseteq R_C$ that the service provider plays, thereby yielding the service boundary behaviour of these roles. To do this, we use the abstraction operator. The abstraction operator σ: $Nets \times \wp \boldsymbol{L} \rightarrow Nets$, where $\boldsymbol{L}$ is the set of all possible labels, delivers for a given Petri net $\boldsymbol{P}$ and a given set of labels A the

Petri net in which all transitions with a label from A are labelled with the silent label τ.
$\sigma(\boldsymbol{P}, A) = \{P_{\boldsymbol{P}}, T_{\boldsymbol{P}}, F_{\boldsymbol{P}}, \{(t, l) \mid (L_{\boldsymbol{P}}(t) \in A \wedge l = \tau) \vee (L_{\boldsymbol{P}}(t) \notin A \wedge l = L_{\boldsymbol{P}}(t))\}, M_{\boldsymbol{P}}\}$

As an example, consider the choreography C from figure 10. The send and receive events of interactions between the bank and the insurance role from this choreography are: $IN_{\boldsymbol{C}}(\{Bank, Insurance\}) = \{sendRiskDetails, recvRiskDetails, sendInsuranceDetails, recvInsuranceDetails\}$. Hence, to derive the service boundary behaviour of the bank and the insurance role, we have to abstract from these events. To do this, we use the formula: $\sigma(C, IN_{\boldsymbol{C}}(\{Bank, Insurance\}))$. This formula leads to the choreography design from figure 11. The figure clearly shows that the interactions between the bank and insurance role are silent.

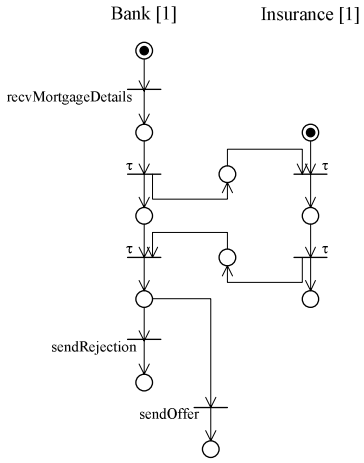

Figure 11. A Restricted Choreography with Silent Transitions

Hence, in the context of a choreography C, the service boundary behaviour of a set of roles $rs \subseteq R_C$ is represented by the Petri net:
$\sigma(mark\ (\cup_{r \in rs} M^0_r)\ (N_C \setminus (\cup(map\ subnet\ rs) \cap T_{\boldsymbol{C}})), IN_C(rs))$

For a provider behaviour P to display at least the same behaviour as a choreography C, P must at least be able to engage in the same events that C can engage in. We say that P must be able to engage in all the same events in which C can engage, because a choreography defines a standard. Therefore, although P may be able to perform its work without engaging in some events, the choreography defines it necessary that P does. As an example, consider that a choreography specifies that a loan agency must inform a regulative authority of all loans that it provides. It does this by specifying that the 'loan agency' role sends 'loan information' to the 'regulative authority' role. Obviously a service provider that wants to play the 'loan agency' role must then also send the 'loan information'.

Figure 9 illustrated that a service provider can be involved in more than one choreography. Therefore, a single choreography may not cover all events of a service provider. Instead, that service provider may also engage in events that are covered by other choreographies. Hence, to compare a service provider to a *single* choreography, we must abstract from the events that are covered by other choreographies. We do this by abstracting from such events, using the abstraction operator that we introduced above. In addition to this, service providers may specify events that are not used in any choreography, but that may be used in the future or may have been used in the past. These events must be removed from the service provider, before comparing it to any choreography. Removing events differs from abstracting from events, because abstracted events still happen but are invisible to the observer, while removed events do *not* happen. Hence, the removal operator ρ: $Nets \times \wp \boldsymbol{L} \rightarrow Nets$, delivers for a given Petri net $\boldsymbol{P}$ and a given set of labels A the Petri net in which all transitions with a label from A are removed, as well as the flow relations and labels that contain these transitions.

Formally: $\rho(\boldsymbol{P}, A)$ is the Petri net for which:
$P_{\rho(\boldsymbol{P}, A)} = P_{\boldsymbol{P}}$
$T_{\rho(\boldsymbol{P}, A)} = T_{\boldsymbol{P}} - \{t \mid L_{\boldsymbol{P}}(t) \in A\}$
$F_{\rho(\boldsymbol{P}, A)} = F_{\boldsymbol{P}} \cap (T_{\rho(\boldsymbol{P}, A)} \times P_{\rho(\boldsymbol{P}, A)} \cup P_{\rho(\boldsymbol{P}, A)} \times T_{\rho(\boldsymbol{P}, A)})$
$L_{\rho(\boldsymbol{P}, A)} = L_{\boldsymbol{P}} \cap (T_{\rho(\boldsymbol{P}, A)} \times \boldsymbol{L})$
$M_{\rho(\boldsymbol{P}, A)} = M_{\boldsymbol{P}}$

Hence, to compare provider behaviour P to a choreography C, we have to abstract from events in P that occur in the context of another choreography and remove events from P that do not occur at all. Given a set of events $A \subseteq E_P - E_C$ that occur in another choreography and a set of events $B \subseteq E_P - E_C$ that do not occur, such that $A \cap B = \emptyset$, the behaviour that we want to compare to the choreography is: $\rho(\sigma(N_P, A), B)$

This behaviour must be equal to the service boundary behaviour that we established in the first step. We use the notion of branching bi-similarity (Glabbeek & Weijland 1996) between Petri nets to verify whether two behaviours are equal. Informally, a Petri net $\boldsymbol{P}$ is branching bi-similar to a Petri net $\boldsymbol{Q}$ (notation: $\boldsymbol{P} \sim \boldsymbol{Q}$) if, at any time, a transition in $\boldsymbol{P}$ can be mirrored by a transition in $\boldsymbol{Q}$ and vice versa, while possibly some silent transitions occur before and after the mirrored transition.

Summarizing, we say that a provider behaviour, must be equal to the service boundary behaviour of the roles that the provider fulfils in a choreography, after: (i) we abstracted from the events that occur in the context of another choreography; and (ii) we removed the events that do not occur, while we only consider a single instance of the roles that the provider fulfils.

Formally, we define this relation between a choreography C and a provider behaviour P that plays the roles $rs \subseteq R_C$ of the choreography and that engages in other choreographies to perform the events from $A \subseteq E_P - E_C$ and that does not perform the events from $B \subseteq E_P - E_C$, such that $A \cap B = \emptyset$, as:
$\sigma(\mathit{mark}\ (\cup_{r \in rs}\ M^0_r)\ (N_C \setminus (\cup(\mathit{map\ subnet\ rs}) \cap T_{\boldsymbol{C}})), IN_C(rs)) \sim \rho(\sigma(N_P, A), B)$

As an example, consider the provider behaviour from figure 12 and verify that this behaviour is branching bi-similar to the restricted choreography from figure 11, in which the interactions between the bank and insurance company are abstracted from.

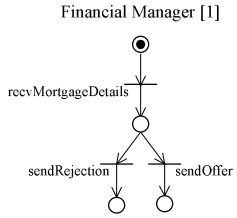

Figure 12. A Provider Behaviour

4.2.2 Derive the Choreography of a Number of Service Providers

We can derive a choreography of a number of service providers, by joining their individual provider behaviours. To join the provider behaviours of service providers, we must first determine the set of interactions IC in which they engage together and the send and receive events that these interactions consist of. We then add these interactions to the joint behaviour of these service providers. The choreography C is then formed as the union of each of the concepts and relations of the provider behaviours P_1, P_2, P_n. For example,

$R_C = R_{P1} \cup R_{P2} \cup \ldots \cup R_{Pn}$, $E_C = E_{P1} \cup E_{P2} \cup \ldots \cup E_{Pn}$, $event_C = event_{P1} \cup event_{P2} \cup \ldots \cup event_{Pn}$. The interactions of the choreography are formed by the set of interactions that were determined by the designer:
$I_C = IC$. Verify that there can be two or more service providers that fulfil the same role, which leads to multiple different instances of that role. This situation can be catered for under the assumption that when two service providers SP^1 and SP^2 fulfil the same role, their respective designs must be equal and their role instances must be different: $RI_{SP1} \cap RI_{SP1} = \varnothing$.

The behaviour of the choreography is the Petri net that is the join of the Petri nets of the provider behaviours. Again we assume that, if two service providers fulfil the same role, their Petri nets are structurally equivalent (structurally equivalent, meaning that they are equivalent under the equals sign (=) rather than the notion of branching bi-similarity). The Petri net that represents the joint behaviour of the service providers consists of the original Petri nets that represent the behaviours of the service providers, the places that correspond to the interactions that these service providers have with each other and the flow relations that relate the interactions to the send and receive events that constitute them. If two service providers perform the same role, we assume that their Petri nets are structurally equivalent. Hence, their places, transitions, flow relations and labelling functions are duplicates of which we only need one. However, because they represent two distinct instances of the same role, their labelling function should be duplicated.

Formally, we define the join operator on two Petri nets $\oplus$: $Nets \times Nets \times \boldsymbol{I} \rightarrow Nets$, where $\boldsymbol{I}$ is the set of all possible interactions, such that for Petri nets N and M and the set of interactions IC, the joint behaviour $N \oplus_{IC} M$ is the Petri net where:
$P_{N \oplus ICM} = P_N \cup P_M \cup IC$
$T_{N \oplus ICM} = T_N \cup T_M$
$F_{N \oplus ICM} = F_N \cup F_M \cup \{(t, (se, re)) \mid (se, re) \in IC \wedge t \in T_{N \oplus ICM} \wedge L(t) = se\} \cup$
$\{((se, re), t) \mid (se, re) \in IC \wedge t \in T_{N \oplus ICM} \wedge L(t) = re\}$
$L_{N \oplus ICM} = L_N \cup L_M$
$M_{N \oplus ICM} = M_N \oplus M_M \cup (IC \times \{0\})$
Where $\oplus$: $\boldsymbol{M} \times \boldsymbol{M} \rightarrow \boldsymbol{M}$ is the function that duplicates markings if necessary. It does this by adding up the number of tokens for places that occur in both markings. Formally:
$(p, i + j) \in M_1 \oplus M_2$ if and only if $(p, i) \in M_1$ and $(p, j) \in M_2$
$(p, i) \in M_1 \oplus M_2$ if and only if $(p, i) \in M_1$ and $(p, j) \notin M_2$
$(p, j) \in M_1 \oplus M_2$ if and only if $(p, i) \notin M_1$ and $(p, j) \in M_2$

Without proof, we claim that the join operator is commutative and associative and that therefore, the Petri net that specifies the behaviour of the choreography can be defined as follows:
$N_C = N_{P1} \oplus_{IC} N_{P2} \oplus_{IC} \ldots \oplus_{IC} N_{Pn}$

As in section 4.2.1 the service providers must *at least* engage in the same tasks and interactions as the choreography, but may implement other tasks and interactions as well. Therefore, after constructing the joint behaviour of the service providers, we may remove events in which the service providers do not engage and we can abstract from events in which the service providers engage, but not in the context of this choreography. To do this, we can use the operators that are defined for this purpose in section 4.2.1.

4.3 Relating Interface Behaviours to Provider Behaviours

Looking at the goals with which designs are constructed from the provider behaviour viewpoint and the interface behaviour viewpoint, we can say that the events from an interface behaviour are a subset of those from a provider behaviour. While a provider behaviour

describes *all* events that a service provider engages in with its environment, an interface behaviour focuses on the events that a service provider engages in with particular business partners, to achieve a certain goal.

Although it would be logical to assume that the business partner and goal that an interface behaviour deals with are the same as the ones in a choreography, this is not necessarily the case. The reason for this is that an interface behaviour may have been designed independently of a particular choreography and therefore with a different goal and different business partners in mind. However, if an interface has been designed specifically to support a choreography, it is best that its goal matches the goal of the choreography and the business partners that it deals with are identified by a particular role.

A provider behaviour can be completely partitioned into a set of interface behaviours. By this we mean that the interface behaviours together must engage exactly in the events of the provider behaviour and no two interface behaviours may engage in the same event. An interface behaviour can not define any events that are not also in the provider behaviour of the service provider to which it belongs, because by definition the provider behaviour of the service provider must completely specify the externally observable behaviour of that service provider. If a provider behaviour does not include certain events that are defined in one of the provider's interface behaviours, then this definition is violated. Also, the provider behaviour can not describe any events that do not occur in one of the interface behaviours of the service provider. The reason for this lies in that interactions with business partners always occur at particular locations (i.e. endpoints). In a design we represent the behaviour at a particular location with an interface. Therefore, each interaction and task must be assigned to an interface. Hence, formally, if a provider behaviour P is partitioned into the interface behaviours $I_1, I_2, \ldots, I_n$ then the set of events of the provider behaviour equals the union of the sets of events of the interfaces and the interfaces must not have any events in common:
$E_P = E_{I1} \cup E_{I2} \cup \ldots \cup E_{In}$
$1 \leq i, j \leq n \wedge i \neq j \Rightarrow E_{Ii} \cap E_{Ij} = \varnothing$

Hence, an interface behaviour must be equal to a provider behaviour after abstracting from the tasks that do not occur in the interface behaviour. Formally, for an interface behaviour I and a provider behaviour P, we denote this as:
$N_I \sim \sigma(N_P, E_P - E_I)$

As an example, we can see that the two interface behaviours from figure 6 conform to the provider behaviour from figure 7.

The partitioning of a provider behaviour into several interface behaviours is lossy, because the joint behaviour of the interface behaviours allows for more freedom than the provider behaviour from which it originates. This means that certain flow relations, silent transitions and places, are lost in the partition. The reason for the loss of relations in the partition of a provider behaviour is that each interface behaviour only captures the relations between its own events. Therefore, the relations between events from different interface behaviours are lost when partitioning the provider behaviour. This can be seen in the provider behaviour from figure 6, in which the interfaces of the service provider are indicated by rounded rectangles. In this figure it can be seen that there are three flow relations and one place that are outside these interfaces and that, therefore, are lost in the partition from figure 7. Hence, the provider behaviour is equal to the union of the interface behaviours *and* the behaviour that describes the relations between different interface behaviours.

Formally, for a provider behaviour P that is partitioned into the interface behaviours $I_1, I_2, \ldots, I_n$ and the Petri net NI that describes the flow relations between events from different interface behaviours, we define:
$N_P = N_{I1} \cup N_{I2} \cup \ldots \cup N_{In} \cup NI$

As an example, figure 13 shows the Petri net that represents the relations between the events of the interfaces from figure 7. The transitions from figure 13 are the transitions from figure 7 that have the same label. The lower place from figure 13 is the place from figure 7 that has {*receiveOffer*} as its pre-set. The union of the interface behaviours from figure 6 with the inter-interface behaviour from figure 13 yields the provider behaviour from figure 7. Note that the Petri net from figure 13 can not be used for anything but to relate the interfaces of the provider behaviour, because it does not do anything itself. This is because this Petri net only represents additional *constraints* that further restrict the behaviour that is constituted by the joint interface behaviours.

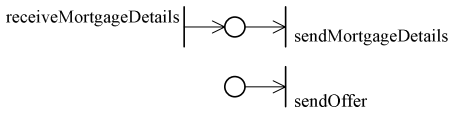

Figure 13. Inter-Interface Constraints

The union operator on Petri nets $\cup$: $Nets \times Nets \rightarrow Nets$ is trivially defined as the union of the individual properties of the Petri nets. *NI* must not define any new events, but only relate events that are defined in the interfaces:
$map\ L_{NI}\ T_{NI} \subseteq E_{I\,1} \cup E_{I\,2} \cup \ldots \cup E_{I\,n} \cup \{\tau\}$

4.4 Relating an Orchestration to Interface Behaviours

An orchestration is intended to realize part of the provider behaviour of a service provider. More specifically, it specifies what a service provider does internally to realize the service that it provides to a particular business partner. The provided service is represented by the interface meant for that business partner. The realization includes internal tasks that must be performed by the service provider and events that are related to interactions that the service provider may have with other business partners in order to realize the communication pattern.

Hence, the behaviour of the orchestration must be exactly equal to the interface behaviour that represents the service that it realizes, after abstracting from the internal tasks and events that are outside that interface behaviour. Formally, for an orchestration *O* and an interface behaviour *SI* that represents the service that the orchestration implements:
$N_{SI} \sim \sigma(N_O, IT_O \cup (E_O - E_{SI}))$

An orchestration may engage in interactions on other interfaces to realize a service. These interfaces represent services that the service provider requires of other service providers. The behaviour of the choreography must be equivalent to the behaviour at each of these interfaces, after abstracting from the events in which the service provider does not engage at that particular interface. Formally, for an orchestration *O* and an interface behaviour *RI* that represents a service that the orchestration uses:
$N_{RI} \sim \sigma(N_O, IT_O \cup (E_O - E_{RI}))$

5. Related Work

In this section, we provide an overview of previous work in two areas related to our proposal, namely: (i) languages for modelling various aspects of services and in particular choreographies and orchestrations; and (ii) definition of relations between service modelling viewpoints.

5.1 Service Modelling Languages

A number of languages for specifying service choreographies (also called *conversations* by some authors[1]) have been proposed in the literature. Benatallah et al. (2003) formulate some requirements for service conversation modelling languages, a service conversation in their terminology being a two-party choreography involving a requester and a provider. The requirements include genericity, automated support, and relevance. The authors argue that state machines satisfy these requirements and sketch an architecture of a service conversation controller capable of monitoring messages exchanged between a requester and provider in order to determine whether they conform to a conversation.

The idea of using state machines for specifying service interactions is also advocated by Bultan et al. (2003) and Hull et al. (2003), who study the formal expressiveness of service choreography (or conversation) specification languages based on communicating state machines (and more specifically Mealy machines). Expressiveness is defined in terms of sets of recognised message traces. The authors put forward sets of message traces which cannot be recognised by isolated state machines while they can be recognised by collections thereof. Frølund & Govindarajan (2003) also adopt a trace-oriented approach to service modelling by proposing a language for specifying (two-party) choreographies based on typed traces: regular expressions without iteration in which the tokens correspond to typed and directed message exchanges.

A number of alternative languages for service orchestration modelling have been put forward by several authors. Benatallah et al. (2003) for example propose a system for model-driven service orchestration called SELF-SERV, in which service orchestrations are modelled using statecharts. Casati et al (2001) on the other hand promote the use of flow charts for this purpose, while DAML-S (2002) advocates the use of preconditions and postconditions expressed in a logic-based language for specifying service processes (which correspond to orchestrations) as well as service interfaces (which correspond to interface behaviours).

Our contribution differs from the work cited above in two ways. First, we distinguish four different viewpoints, whereas the above efforts deal with either one or two of these viewpoints at a time. More importantly, we recognise and address the need to formally capture relationships between viewpoints as a step to enable incremental and collaborative service design and verify global design consistency. In the next sub-section, we look at other related work in which the relationship between viewpoints is considered.

5.2 Relating Service Modelling Viewpoints

The choreography vs. orchestration dichotomy is informally discussed and illustrated by Peltz (2003) who analyses the overlap and complementarity of three proposed web service composition standards, namely WSCI, BPML, and BPEL4WS, with respect to this dichotomy. Interestingly, Peltz does not make the distinction between choreographies, interface behaviours, and provider behaviours. Instead, Peltz considers that interface behaviour (e.g. a BPEL4WS abstract process) is the same concept as choreography. In this paper, we have shown that by making this distinction, it is possible to enable an incremental approach to service consistency verification, wherein interface and provider behaviours (defined independently by the service providers) are validated against choreographies (defined by agreement between multiple providers or by standardisation committees).

---

[1] The term *service conversation* seems to originate from the area of agent systems, where a number of conversation protocols and languages have been developed – see e.g. standardisation work by the FIPA organisation (www.fipa.org). We prefer the term *choreography* mainly to oppose it to *orchestration*.

Mecella et al. (2001) advocate the use of Petri nets for specifying service choreographies (which the authors call *orchestration net*s). These choreographies are intended to be enacted by a number of partners cooperating through so-called "cooperative gateways". On the basis of this approach, the authors define a set of formal conditions to verify whether a service can be replaced by another (substitutability) in a given service choreography. This is similar to checking whether a given provider behaviour conforms to a choreography in our approach.

Wombacher & Malheko (2002) address the issue of comparing required behavioural interfaces with provided behavioural interfaces for service matchmaking. The authors assume that interface behaviours are specified as statecharts. Their approach proceeds by computing the "perspective" of the provider on the requestor's interface (using a "mirroring" operator on the underlying statechart), and then determining whether this "perspective" is a subset of the provided interface. This differs from our approach in which the relation between the interface required by a given partner and the interface provided by another is made with respect to a given choreography.

The orchestration and interface behaviour modelling viewpoints have also been investigated in the areas of B2B Integration (Bussler 2003) and inter-organizational workflow (Grefen et al. 2000, Aalst & Weske 2001, Aalst 2002). In these areas, a separation is often made between the notions of *private process* (corresponding to orchestration in our approach) and *public process* (corresponding to interface behaviour, provider behaviour, and choreography in our approach). Especially related to our work is the Public-to-Private (P2P) approach of Aalst & Weske in which choreographies (called public workflows) and orchestrations (private workflows) are modelled using WF-Nets: a class of labelled Petri nets. These viewpoints are then related through formal notions of workflow inheritance and the soundness of the overall inter-organisational workflow is ensured by checking the consistency between the orchestrations and the subset of the choreography that they "implement". Our work can be seen as an extension of that of Aalst & Weske, in which the notions of interface behaviour and provider behaviour are explicitly introduced to complement those of choreography and orchestration. Introducing these intermediate notions enables a more incremental approach to service-oriented design.

6. Conclusion

The paper has introduced a core model for service-oriented design upon which different design viewpoints can be defined. Four such viewpoints have been identified and formalised, as have a number of relations between these viewpoints. These relations can be used to perform (global) consistency checking of multi-viewpoint service designs thereby providing a formal foundation for incremental and collaborative approaches to service-oriented design. Specifically, choreographies designed by agreement between multiple parties (or by industry-specific consortia) can be checked against provider behaviours defined independently by each of the involved parties. Then, each provider can check her internal orchestration designs against her provider behaviours. Also, pairs of behaviour interfaces describing the interactions between any pair of providers involved in a given choreography can be checked for conformance against each other.

We have implemented a proof-of-concept tool that supports the relational operators defined in section 4 (see screenshot in Figure 14). The tool allows a designer to create a project to which designs from the four supported viewpoints can be added. For example, the screenshot shows a project containing a choreography design and a provider behaviour. When a project contains multiple designs, the tool allows the designer to check the relations between them. To do so, the designer can enter algebraic expressions similar to the ones presented in section 4. For example, the figure shows how a designer checks whether a provider behaviour is equivalent to part of a choreography. According to the tool, this is 'true'. The tool can be downloaded

from (Dijkman 2004), along with the example from the figure and an explanation of that example.

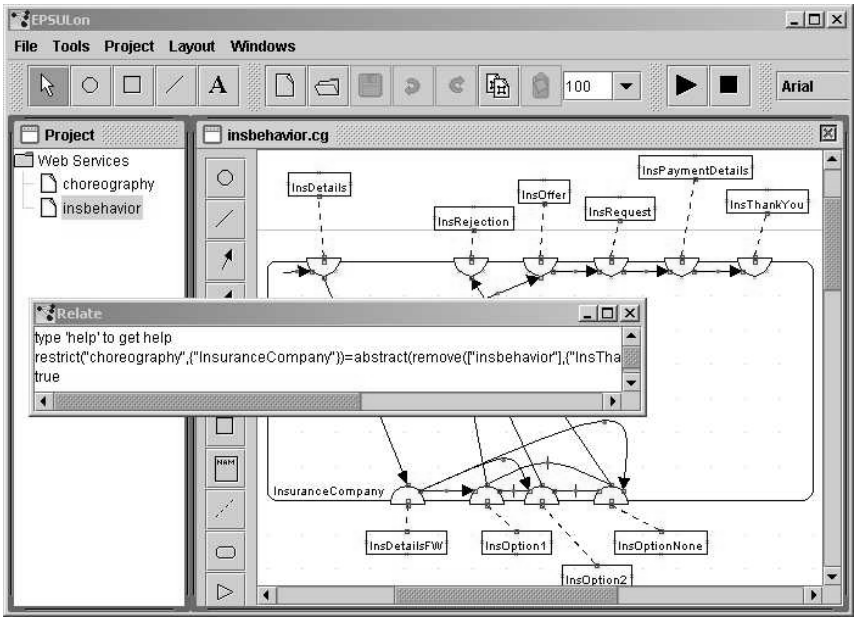

Figure 14. A Screenshot of the Tool

The tool also illustrates a concrete approach to Service-oriented Design based on the core model introduced in section 2, but using a more designer-oriented language than Petri nets. Specifically, the approach relies on a graphical notation called Interaction Systems Design Language (ISDL) originally intended for architecture description. ISDL is discussed in various papers (Quartel et al. 2002, Quartel 1998, Quartel et al. 1997) and a tutorial can be found at (Quartel n.d.). Motivating the use of ISDL as a language for service-oriented design is out of the scope of this paper.

Currently, the tool uses a naïve algorithm for checking the relations between viewpoint designs and adopts a trace-based semantics. Specifically, all possible traces of the designs to be verified are generated and consistency is checked by comparing sets of traces. Trace-based semantics is known to be less expressive than a Petri net-based semantics with bi-simulation equivalence (Glabbeek 2001). At the same time, under suitably chosen restrictions, the complexity of verification of bisimulation equivalence in Petri nets can be manageable and even considerably lower than the one of the naïve algorithm used in the tool. Specifically, bi-similarity of Petri nets is decidable for situations in which at least one of the Petri nets is deterministic (up to bi-similarity) (Jančar 1994). Moreover, bi-similarity between two Petri nets is known to be decidable in polynomial time for bounded Petri nets (Alvarez et al. 1991). Therefore, if we restrict our behavioural semantics to deterministic bounded Petri nets, a reasonably efficient algorithm can be constructed to verify the conformance between the viewpoints. We feel that the imposed restriction is reasonable as current service description languages are based on constructs that can be mapped to bounded Petri-nets, which Martens (2003) shows for BPEL4WS.

An obvious direction for future work is thus to investigate the application to our approach, of consistency checking algorithms based on bi-simulation equivalence or other related notions of behavioural equivalence. In particular, adapting some of the results from the area of inheritance of (Petri net-based) workflows (Aalst 2002) is an appealing alternative.

Acknowledgments. The authors wish to thank Arthur ter Hofstede for his significant feedback on earlier drafts of this paper.

References

Aalst, W. van der (2002). Inheritance of interorganizational workflows to enable business-to-business e-commerce. *Electronic Commerce Research*, *2*(3), 195–231.

Aalst, W. van der (2003). Don't go with the flow: Web services composition standards exposed. *IEEE Intelligent Systems*, *18*.

Aalst, W. van der, & Weske, M. (2001). The P2P approach to interorganizational workflows. *Lecture Notes in Computer Science*, *2068*, 140–155.

Alonso, G., Casati, F., Kuno, H., & Machiraju, V. (2003). *Web services: Concepts, architectures and applications*. Springer Verlag.

Alvarez, C., Balcazar, J., Gabarro, J., & Santha, M. (1991). Parallel complexity in the design and analysis of concurrent systems. *Proc. of the Conf. on Parallel Architectures and Languages Europe (PARLE)*, no. 505. in Lecture Notes in Computer Science: Springer Verlag.

Arkin, A., et al. (2002a). Business process modeling language. Retrieved January 29, 2004, from BPMI Website: http: //www.bpmi.org/bpml-spec.esp.

Arkin, A., et al. (2002b). Web services choreography interface (WSCI) 1.0. Retrieved January 29, 2004, from World Wide Web Consortium Website: http://www.w3.org/TR/2002/NOTE-wsci-20020808.

BEA Systems, Microsoft, IBM, & SAP (2003). Business process execution language for web services version 1.1. Retrieved January 29, 2004, from IBM Website: http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/.

Benatallah, B., Dumas, M., Sheng, Q., & Ngu, A. (2002, February). Declarative composition and peer-to-peer provisioning of dynamic web services. *Proc. of the 18th IEEE Intl. Conf. on Data Engineering (ICDE)*. San Jose CA, USA: IEEE Press.

Benatallah, B., Sheng, Q., & Dumas, M. (2003). The Self-Serv environment for web services composition. *IEEE Internet Computing*, *7*(1), 40–48.

Benattallah, B., Casati, F., Toumani, F., & Hamadi, R. (2003). Conceptual modeling of web service conversations. *Proc. of the 15th Intl. Conf. on Advanced Information Systems (CAiSE)*, no. 2681 in Lecture Notes in Computer Science (pp. 449–467). Klagenfurt, Austria: Springer Verlag.

Bultan, T., Fu, X., Hull, R., & Su, J. (2003, May). Conversation specification: A new approach to design and analysis of e-service composition. *Proc. of the Intl. Conf. on the World Wide Web (WWW)* (pp.403–410). Budapest, Hungary: ACM Press.

Bussler, C. (2003). *B2B integration - concepts and architecture*. Springer Verlag.

Casati, F., & Shan, M.-C. (2001). Dynamic and adaptive composition of e-services. *Information Systems*, *26*(3), 143–162.

DAML-S Consortium (2001). DAML services. http://www.daml.org/services.

Dijkman, R. (2004). EPSULon for web services. Retrieved January 29, 2004, from, University of Twente, Centre for Telematics and Information Technology Website: http://wwwhome.cs.utwente.nl/~dijkman/EPSULon4WS.html.

Eijk, P. van, Vissers, C., & Diaz, M. (Eds.). (1989). *The formal description technique LOTOS*. Elsevier Science.

Frølund, S., & Govindarajan, K. (2003). *cl: A language for formally defining web services interactions* (Technical Report HPL-2003-208). Hewlett-Packard.

Glabbeek, R. van (2001). *Handbook of process algebra*, Chap. 1: The linear time – branching time spectrum I: The semantics of concrete sequential processes, pp. 3–99. Elsevier Science.

Glabbeek, R. van, & Weijland, W. (1996). Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, *43*(3), 555–600.

Grefen, P., Aberer, K., Hoffner, Y., & Ludwig, H. (2000). CrossFlow: Cross-organizational workflow management in dynamic virtual enterprises. *Intl. Journal of Computer Systems Science & Engineering*, *15*(5), 277–290.

Hohpe, G., & Woolf, B. (2004). *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley.

Hull, R., Benedikt, M., Christophides, V., & Su, J. (2003, June). E-services: A look behind the curtain. *Proc. of the 22nd ACM Symposium on Principles of Database Systems (PODS)* (pp. 1–14). San Diego, CA, USA: ACM Press.

IEEE Architecture Working Group (2000, September). *IEEE recommended practice for architectural description of software-intensive systems* (Technical Report IEEE-Std 1471-2000). IEEE Computer Society.

ITU-T/ISO (1994–1997). *Open distributed processing reference model* (Technical Report ITU-T X.901..904 — ISO/IEC 10746-1..4). ITU-T/ISO.

Jančar, P. (1994, February). Decidability questions for bismilarity of Petri nets and some related problems. In P. Enjalbert, E. Mayr, & K. Wagner (Eds.), *Proc. of the 11th Annual Symposium on Theoretical Aspects of Computer Science* (pp. 581–192). Caen, France.

Martens, A. (2003). *Verteilte geschäftsprozesse – modellierung und verifikation mit hilfe von web services*. PhD thesis, Humboldt Universität, Berlin, Germany.

Mecella, M., Pernici, B., & Craca, P. (2001, September). Compatibility of e -services in a cooperative multi-platform environment. *Proc. of the 2nd Intl. Workshop on Technologies for E-Services*, no. 2193 in Lecture Notes in Computer Science (pp. 44–57). Rome, Italy: Springer Verlag.

Milner, R. (1989). *Communication and concurrency*. Prentice Hall.

Milner, R. (1999). *Communicating with mobile agents: the pi-calculus*. Cambridge University Press.

Pelz, C. (2003). Web services orchestration and choreography. *IEEE Computer*, *36*(8), 46–52.

Quartel, D. (1998). *Action relations - basic design concepts for behaviour modelling and refinement*. PhD thesis, University of Twente, Enschede, The Netherlands.

Quartel, D. (n.d.). ISDL home. Retrieved January 29, 2004, from, University of Twente, Centre for Telematics and Information Technology Website: http://isdl.ctit.utwente.nl/.

Quartel, D., Ferreira Pires, L., & Sinderen, M. van (2002). On architectural support for behaviour refinement in distributed systems design. *Journal of Integrated Design and Process Science*, *6*(1).

Quartel, D., Ferreira Pires, L., Sinderen, M. van, Franken, H., & Vissers, C. (1997). On the role of basic design concepts in behaviour structuring. *Computer Networks and ISDN Systems*, *29*(4), 413–436.

Szyperski, C. (2003). Component technology – what, where, and how? *Proc. of the 25 th Intl. Conf. on Software Engineering (ICSE)* (pp. 684–693). Portland, OR, USA.

UN/CEFACT, & OASIS (2001). ebXML business process specification schema (version 1.01). Retrieved, January 29, 2004 from ebXML Website: http://www.ebxml.org/specs/ebBPSS.pdf.

W3C (2003). Web services description language (WSDL): Part 1: Core language version 1.2. Retrieved January 29, 2004, from World Wide Web Consortium Website: http://www.w3.org/TR/2003/WD-wsdl20-20031110.

Wohed, P., Aalst, W. van der, Dumas, M., & Hofstede, A. (2003, October). Analysis of web services composition languages: The case of BPEL4WS. *Proc. of the 22nd Intl. Conf. on Conceptual Modelling (ER)*. Chicago IL, USA: Springer Verlag.

Wombacher, A., & Mahleko, B. (2002, October). Finding trading partners to establish ad-hoc business processes. *Proc. of the Intl. Conf. on Cooperative Information Systems*, no. 2519 in Lecture Notes in Computer Science. Irvine, CA, USA: Springer Verlag.