Methodological Support for Service-oriented Design with ISDL

Dick Quartel
University of Twente
PO Box 217
7500 AE Enschede
+31 53 4893765
d.a.c.quartel@utwente.nl

Remco Dijkman
University of Twente
PO Box 217
7500 AE Enschede
+31 53 4894454
r.m.dijkman@utwente.nl

Marten van Sinderen
University of Twente
PO Box 217
7500 AE Enschede
+31 53 4893677
m.j.vansinderen@utwente.nl

ABSTRACT
Currently, service-oriented computing is mainly technology-driven. Most developments focus on the technology that enables enterprises to describe, publish and compose application services, and to communicate with applications of other enterprises according to their service descriptions. In this paper, we argue that this technology should be complemented with modelling languages, design methods and techniques supporting *service-oriented design*. We consider service-oriented design as the process of designing application support for business processes, using the service-oriented paradigm. We assume that service-oriented computing technology is used to implement application support. The paper presents two main contributions to the area of service-oriented design. First, a systematic service-oriented design approach is presented, identifying generic design milestones and a method for assessing the conformance between application designs at related abstraction levels. Second, a conceptual model for service-oriented design is presented that provides a common and precise understanding of the terminology used in service-oriented design. The ISDL modelling language is introduced to express service-oriented designs, based on this conceptual model. The paper includes an elaborate example to illustrate our ideas.

Categories and Subject Descriptors
D.2.1 [**Software Engineering**]: Requirements/Specifications – *Methodologies*; H.1.1 [**Models and Principles**]: Systems and Information theory – *General systems theory*.

General Terms
Design, Languages, Verification.

Keywords
Service-oriented design, service-oriented computing, ISDL, service modelling, service composition.

ICSOC'04, November 15–18, 2004, New York, New York, USA.

1. INTRODUCTION
Enterprises form and change business partnerships during their lifetime. For example, a business process may be out-sourced for efficiency reasons, or different processes may be integrated to provide a new product. In addition, enterprises increasingly use software applications to support their business processes. One can conclude from these observations that there is a growing need for linking software applications to support business partnerships.

Service-oriented computing promises to deliver the methods and technologies to help business partners to link their software applications. This should facilitate the introduction of richer and more advanced applications, thereby offering new business opportunities. Other foreseen benefits are the shortening of application development time by reusing available applications, and the creation of a service market, where enterprises make it their business to offer generic and reusable services that can be used as application building blocks.

Informally the service-oriented paradigm is characterized by the explicit identification and description of the externally observable behaviour, or *service*, of an application. Applications can then be linked, based on the description of their externally observable behaviour. According to this paradigm, developers do in principle not need to have any knowledge about the internal functioning of the applications being linked.

Currently, service-oriented computing is mainly technology-driven. Most developments focus on the technology that enables enterprises to describe the services they offer in a textual, mostly XML-based, form (e.g.: [29], [30]), to publish these descriptions on-line and find services of other enterprises according to these descriptions (e.g.: [26]), to compose services into new services (e.g.: [5], [7]), and to communicate with applications of other enterprises according to their service descriptions (e.g.: [28]). We argue that, as in other areas of computing, this technology should be complemented with modelling languages and methods supporting *service-oriented design*. We consider service-oriented design as the process of designing application support for one or more business processes, using the service-oriented paradigm.

The contribution of this paper is twofold. First, a service-oriented design approach is presented. This approach identifies generic milestones in the process of designing application support for business processes that can be implemented using service-oriented computing technology. In addition, the approach describes a method to assess the conformance between designs defined at different, but

related, abstraction levels. Second, the paper presents a conceptual model that provides a common and precise understanding of the terminology that is used in service-oriented design (and computing). The Interaction System Design Language (ISDL) is introduced to express the concepts from this conceptual model. This modelling language supports our service-oriented design approach, by allowing one to express the milestones and assess the conformance between them.

This paper is further structured as follows. Section 2 provides an overview of service description and composition techniques being used in service-oriented computing. Section 3 explains our service-oriented design approach. Section 4 presents a conceptual model for service-oriented design, and introduces ISDL to express service-oriented designs based on this conceptual model. Section 5 illustrates the application of ISDL in our service-oriented design approach with an example. And section 6 concludes this paper.

2. SERVICE-ORIENTED COMPUTING

In this section we look at service description and composition languages in more detail. Service description languages are used to represent relevant properties of services, and service composition languages provide techniques to compose a service from other services. These languages are relevant from a design perspective, because in the end a service-oriented design has to be mapped onto the description and composition languages offered by service-oriented computing technology.

2.1 Service description

A service description specifies the externally observable behaviour of an application. This defines the way in which an application can be used by another application. We distinguish two levels of service description in service-oriented computing: interface description and interface behaviour description.

An interface description specifies the individual interactions that a service can have with its environment. Different description techniques imply different mechanisms for interaction, such as request-response and one-way message passing. However, all description techniques agree that the basic mechanism for interaction is one-way message passing and define their more complex interaction mechanisms as one-way message passing patterns [14]. Hence, an interface description implicitly defines the messages that a service is ready to receive and the messages that it may send. The description languages define the relation between an interface description and the concrete syntax of the messages that can be exchanged by the service. Hence, the interface description is sufficient to allow users to interact with the service. In addition, interface description techniques allow for logical grouping of message send and receive events, in terms of messaging patterns such as the ones described above and in terms of groupings of these patterns.

An interface behaviour description specifies the possible orders in which messages can be sent and received by a service. Examples of interface behaviour description languages are BPEL4WS abstract processes [7] and WSCI [29]. Interface behaviour descriptions provide service users with more information about how to use the service. These behaviour descriptions can also be used to verify at run-time whether the service behaves according to its behaviour description. Interface behaviour description techniques draw on description techniques for business processes and use many of the patterns these description techniques use [1], [32]. Like business process description techniques they distinguish (business) tasks that can, for example, be composed in sequence, parallel or choice. They consider sending and receiving messages as special forms of tasks.

2.2 Service composition

Service composition descriptions describe the way in which application services use each other. We distinguish between two forms of service composition description: choreography and orchestration description (also see [8], [22]).

A choreography describes the interactions that two or more applications have with each other to achieve a common goal, and the relations between these interactions. Therefore, the logic that executes a choreography must be distributed over the application service providers. A typical example of a choreography description language is the web-services choreography model [31]. Choreography descriptions can serve different purposes. They can be used as standard business processes in which application service providers can indicate the parts that they can fulfil. Then, the providers can use these descriptions as a basis to start implementing their services. Alternatively, choreography descriptions can be executed by choreography engines, such as [10], [18], which manage the interactions between the right providers and in the correct order.

An orchestration describes the interactions that a single application service provider has with other providers to provide its own service. Hence, unlike in a choreography, the interactions in an orchestration focus on a single provider. Therefore, these interactions can be directly executed by that provider. Typical examples of orchestration description languages are BPEL4WS executable process [7] and BPML [5]. Orchestrations can be executed by a so-called orchestration engine, much like business processes can be executed in workflow engines.

Like interface behaviour description languages, service composition description languages draw on languages for business process description to describe the relations between their interactions.

3. SERVICE-ORIENTED DESIGN

The purpose of service-oriented design is to systematically design application support for business processes, which is being implemented using service-oriented computing technology. For example, multiple design steps producing multiple related designs may be required to translate business requirements into the facilities provided by some service-oriented computing technology. Furthermore, service-oriented design is required to distinguish between technology independent and technology dependent service models, as being advocated by the model-driven architecture approach of OMG [20].

We claim that our service-oriented design approach is generally applicable to distributed information systems. Therefore, we also use the term system instead of enterprise or application in the sequel. Furthermore the principles of service-oriented design are not new [27]. The emergence of service-oriented computing, however, facilitates the mapping of service-oriented designs onto service-oriented computing technology, thereby allowing one to follow the service-oriented paradigm throughout the entire development process.

3.1 The role of service in system design

The Merriam-Webster dictionary defines a system as

> *a regularly interacting or interdependent group of items forming a unified whole.*

This definition is of interest because it distinguishes two different system perspectives: an internal perspective, which is referred to as the "interacting or interdependent group of items", and an external perspective, which is referred to as the "unified whole".

The *external system perspective* corresponds to the perspective of the system users. These users are only interested in the functionality, or behaviour, provided by the system as a whole, and not in how the system is internally constructed. The system is considered as a black box, and the externally observable behaviour of the system is called the system's *service*. This service can be defined as the set of possible interactions between the system and its environment (the service users) that the system is capable of supporting, including the possible relationships between these interactions.

The *internal system perspective* corresponds to the perspective of the system designers. The definition expresses that the unified whole, as seen and experienced by the users, actually does not exist as a single, monolithic entity, but is formed by a group of interdependent items, or system parts. The internal perspective shows how the system is internally structured as a composition of parts. These parts have to interact amongst each other to fulfil the purpose of the system as a whole.

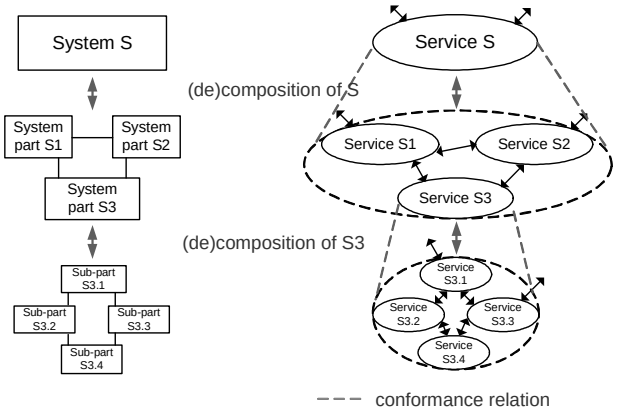


Figure 1. External and internal system perspectives

By considering each part as a system, the external and internal perspectives can be applied again to the system parts. This results in a process of repeated or recursive decomposition, yielding several levels of decomposition, also called levels of abstraction. Figure 1 depicts this process. The process of recursive decomposition shows that the system concept can represent various kinds of entities, such as applications, collections of communicating applications, enterprises or value chains. Consequently, the service concept as it is used in service-oriented *design* can represent the service provided by various kinds of concrete entities. However, the service concept as it is used in service-oriented *computing* always represents an application service. Therefore, the first is more generic than the latter. As a consequence, it can be used to represent a service that is not (only) implemented in service-oriented technology, but (also) in other technologies or by manual interactions. For example, at an enterprise level an interaction can be implemented by sending a letter or making a phone call. Also, an interaction in a generic service can represent a more abstract interaction that is implemented by a complex pattern of interactions at a lower level of decomposition. For example, at an enterprise level the interaction 'buy item' can exist that is implemented by the interactions 'get item list', 'select item' and 'give customer details' at a lower level of decomposition. The process of recursive decomposition stops when existing system parts are found, e.g., available application services.

Although the term decomposition may suggest a top-down approach, bottom-up design knowledge is necessary to arrive at compositions of available system parts. Typically, one may distinguish the following design activities, or steps, in a decomposition: (i) the definition of the required service, (ii) the proposal of a composition of (available) sub-services, and (iii) checking whether the composition conforms to the required service. In practice, (ii) is largely a bottom-up activity and may precede (i) to quickly obtain a prototype, based on an imprecise idea of the desired service. Such a prototype helps to make up one's mind about the precise characteristics of the desired service, and its implementability.

The trial and error nature of activities (ii) and (iii) imply that alternative compositions may have to be proposed during a design step. Furthermore, in later design steps one may decide to adjust some (composition of) service(s) proposed in an earlier design step, guided by acquired design experience. This gives service-oriented design a cyclic or iterative character.

3.2 Conformance assessment

In a systematic service-oriented design process, we assume that for each design step both the behaviour of the required service and the behaviour of its design in terms of a composition of sub-services, are defined completely. This allows one to assess the conformance between the service specification and its design.

In general, conformance can be obtained in two principally different ways: (i) by following so-called correctness (i.e., conformance) preserving refinement or transformation rules, or (ii) by assessing the conformance of a design afterwards by abstracting from the added design information (see Figure 2). The first approach assumes a strictly top-down approach, and has as advantage that no explicit conformance assessment step is necessary. A disadvantage is however that the applied rules have to be rather specific, prescribing specific (pre-defined) types of compositions for specific types of required services, thereby limiting design freedom.

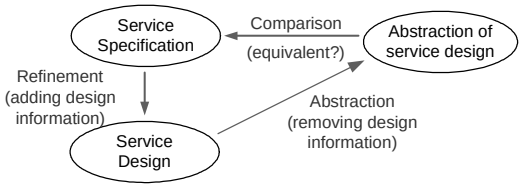


Figure 2. Conformance assessment

The second approach does not prescribe any rules or give any guidance on how the composition is obtained. However, it does allow one to assess the conformance of any proposed service design. This works as follows (see Figure 2). The service design adds design information to the service specification: the interactions between the constituent sub-services, and possibly the refinement of the original service interactions. Hence, to assess conformance, we can abstract from the added design information. After abstracting from this information, the obtained abstraction should be equivalent to the

original service specification. The particular notion of equivalence being applied, determines the type of service refinements (decompositions) that are considered correct. This approach also allows one to derive the service specification from a proposed composition of sub-services, when following a bottom-up design approach.

3.3 Design milestones

A design milestone is the result of one or more design steps, representing a design or specification that satisfies certain design objectives. We consider the following generic design milestones relevant for service-oriented design: business process specification, application service specification, application service design and application service implementation.

3.3.1 Business process specification

The objective of this milestone is to specify the business process that requires application support. This milestone forces a designer to model, analyse and, possibly, redesign the context in which the application must be embedded. Furthermore, the business process defines (indirectly) the business requirements on the desired application support.

In general, different actors may contribute to the activities or tasks performed in a business process, such as clients, administrative workers or software applications. For example, the activity of requesting a hotel reservation via the Web involves the contribution from a client, who provides the reservation information, and a Web application, which validates the information and confirms the request.

In this milestone, we consider each business process activity as a whole, and abstract from the contributions that each of the involved actors may have in this activity. The reason for this is that we want to focus on what the business process should do, and not on how this can be done or by whom. Consequently, this milestone defines the role of a single (virtual) actor that provides the business process as a whole.

3.3.2 Application service specification

The objective of this milestone is to specify the service of the application that must support the business process. This milestone is motivated by the need to specify precisely what functionality is required from the application.

In this milestone the business process is decomposed into a part that is to be supported by the application and a remaining part, called the application environment, which may consist of other applications or human users. This is done by identifying the activities from the business process model in which both the application and its environment are involved. In this way a boundary is determined between the application and its environment, at which they interact through the identified activities, also called interaction activities. This boundary is specified by the application service, which defines the interaction activities to be supported by the application, and their relationships.

In addition, activities may be identified that must be completely supported by the application. From a service perspective, these activities can in principle be ignored since they may unnecessarily constrain the service design. Alternatively, in case the activities are considered relevant, they could be maintained as internal activities and defined as additional requirements on the design.

Figure 3 illustrates the decomposition of a business process into the application and its environment.

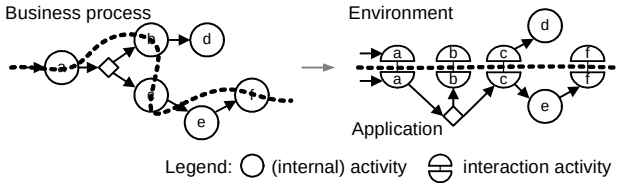


Figure 3. Business process decomposition

3.3.3 Application service design

The objective of this milestone is to design the application service in terms of a composition of sub-services that can be provided by application building blocks. This milestone is needed when no building block is available that completely provides the application service.

Depending on the complexity of the application and the availability of building blocks, multiple design steps as described in section 3.1 may be needed, until one reaches building blocks that are available or can be implemented directly. Observe that this milestone relates the notions of choreography and orchestration: each sub-service defines the orchestration of a single application building block, while the composition of the interactions between the sub-services defines the choreography of the involved building blocks.

This milestone also aims at a service design that is defined independently of any service-oriented computing technology or platform. For this purpose, we assume the existence of an abstract service platform supporting abstract interactions between application building blocks, which can be mapped onto the concrete interactions or interaction patterns supported by middleware technology [3].

3.3.4 Application service implementation

The objective of this milestone is to implement the service design of the previous milestone using a specific service computing technology or platform. This requires one to transform the platform-independent design into a platform-dependent design, using the description and composition techniques provided by the specific service platform. This transformation falls outside the scope of this paper.

4. SERVICE MODELLING

From the analysis of description languages in section 2 and from observations about the service-oriented design process in section 3, we derive a set of concepts that can be used for service-oriented design. This section explains these concepts as well as their graphical representation in the Interaction Systems Design Language (ISDL). It also explains how these concepts can be used for design from the perspective of the milestones from section 3.3.

4.1 Concepts for service-oriented design

The first three milestones from section 3.3 cover both business process design and application service design. Therefore, our concepts, shown in Figure 4, are generalizations of concepts from these domains. Furthermore, according to the fourth milestone, a service-oriented design should eventually be mapped onto implementation related concepts. Therefore, we partly derived the

concepts in Figure 4 from the concepts used in description languages in service-oriented computing. We published earlier versions of the conceptual model in Figure 4 in [8] and [14].

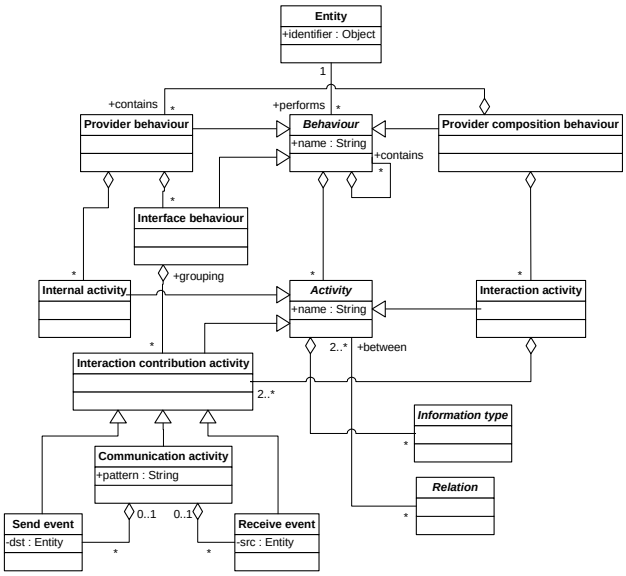


Figure 4. Concepts for service-oriented design

An *entity* represents a system or system part, e.g., a business partner, application or human user. It has a unique identifier, such that it can be addressed. An entity performs some *behaviour*. In general, a behaviour is defined in terms of a collection of related activities. An *activity* represents a logical unit of functionality, e.g., a business task or application function. One may associate one or more *information types* with an activity, representing the type of the result that is established in the activity. We leave it to the particular modelling language how information types and their operations are specified. Therefore, the information type concept is declared abstract.

Relations between activities determine the possible orders in which they can be performed and how the information established by some activity is related to the information established by other activities. Depending on which properties of relations one considers relevant, e.g., only temporal order or also causality, one may use different modelling languages. Therefore, we leave it to a modelling language to define how relations can be specified, and declare the relation concept as abstract. [2] defines a set of relations that are commonly used to specify the possible orders in which activities can be performed.

We distinguish between three types of activities: *internal activities*, *interaction contribution activities* and *interaction activities*. An internal activity represents an activity that an entity performs internally. An interaction activity represents an activity that is performed by multiple entities in cooperation. The contribution of some entity to an interaction activity is represented by an interaction contribution activity. For example, requesting a hotel reservation (section 3.3.1) can be modelled as an interaction activity, which consists of two interaction contribution activities: the contribution of the client entering information regarding the desired reservation, and the contribution of the Web application validating the client input. The processing of this request by the Web application, such as storing the reservation in a database, can be modelled as an internal activity of the Web application.

A *provider behaviour* represents a behaviour that is provided by an entity to its environment. For example, an application service is a provider behaviour, representing the functionality provided by the application to its environment, which consists of the application users. Consequently, a provider behaviour consists of a collection of related interaction contribution activities, and possibly internal activities, since it is associated with a single entity. In case of a pure service definition, only interaction contribution activities are defined. Internal activities are often added to a service definition, however, to represent activities that are considered relevant in understanding and later on designing the relationship between interaction contribution activities. Interaction contribution activities are grouped into *interface behaviours*. For example, different interfaces may be defined to distinguish between interactions with different types of users.

Interaction activities are used to define *provider composition behaviours* that are performed by compositions of entities. An interaction activity is defined by two (or more) interaction contribution activities, representing the interaction or cooperation between the involved provider behaviours. At an abstract level an interaction activity may represent a complex function, e.g., the establishment of a sale. At a concrete level an interaction activity typically represents communication to which entities contribute by performing *communication activities*, which can be as simple as a *send* or *receive event*, representing the sending or receiving of a message, or consist of some pattern of related send and receive events. Since this concrete level is assumed by service-oriented computing technology, these specific types of interaction contribution activities are incorporated in the conceptual model. During service-oriented design, abstract interactions as mentioned above are refined into patterns of paired send and receive events that can be supported by service-oriented computing platforms.

4.2 Representing the concepts in ISDL

The Interaction Systems Design Language (ISDL) [23, 24] is a design language aimed at modelling distributed systems at higher abstraction levels. We used ISDL before for business process and distributed application design [16, 25]. Figure 5 shows how the service-oriented design concepts from Figure 4 can be graphically represented in ISDL. A tutorial on ISDL can be found at [23].

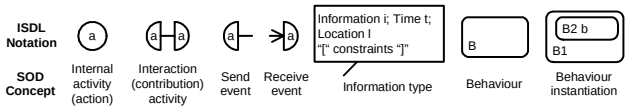
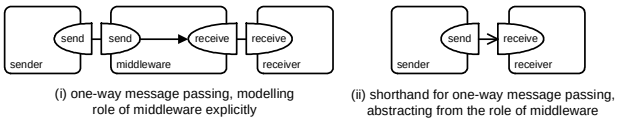


Figure 5. Representation of SO-design concepts in ISDL

ISDL represents internal activities, which it also calls actions, as circles (or ellipses) with the action's name inside it. It represents interaction activities as segments of a circle (or ellipse) that are connected by lines. These segments represent the interaction contribution activities of an interaction activity. ISDL interaction activities are atomic, which means that they either happen for all involved behaviours at the same time, establishing the same result for each behaviour, or that they do not happen at all, in which case no result is established. Consequently, ISDL adopts a synchronous interaction model, requiring entities to be involved in an interaction simultaneously. Although a synchronous interaction can be used to represent one-way message passing from an abstract perspective, it does not consider the passing of time between the moment at which the send event occurs and the moment at which the receive event

occurs. If we want to consider the passing of time, one-way message passing has to be modelled by a synchronous send interaction followed by a synchronous receive interaction. Figure 6 illustrates this, and introduces a shorthand to represent one-way message passing directly in ISDL.



(i) one-way message passing, modelling role of middleware explicitly

(ii) shorthand for one-way message passing, abstracting from the role of middleware

Figure 6. One-way message passing in ISDL

ISDL represents the type of result of an activity inside a box that is attached to the activity. ISDL does not only consider the result of an activity, but also the time moment at which the result is established, and the location at which the result is available. It refers to the result, the time at which the result is established and the location at which it is available as the information, time and location attribute of an activity, respectively. Constraints can be defined on possible values for these attributes. These constraints also specify the relation between attribute values established in different activities. ISDL does not prescribe a language for defining attribute types and constraints, but provides bindings to existing languages that can be used for that purpose. Currently, bindings to the formal description technique Z, to Java and to the functional programming language Q exist.

ISDL uses causality relations to represent the relations between activities. A *causality relation* defines for the associated activity, say a, the causality condition that must be satisfied to enable this activity to happen (occur). This causality condition is defined in terms of three elementary conditions: (i) the *start condition* represents that activity a is enabled from the beginning of some behaviour and independent of any other activity, (ii) *enabling condition* b represents that activity b must have occurred before a can occur, and (ii) *disabling condition* ¬b represents that activity b must not have occurred before nor simultaneously with a to enable the occurrence of a. These elementary conditions can be combined using the *and*- and *or*-operator to represent more complex conditions. Figure 7 depicts some simple examples. In Figure 7(iv) activities b and c are enabled from the beginning (and independent of each other), while action a can only happen after b and c have happened. In Figure 7(v) activity a can happen after activity b or activity c has happened. Figure 7(vi) defines a choice relation between activity a and b, for which a convenient shorthand notation is introduced in Figure 7(vii).

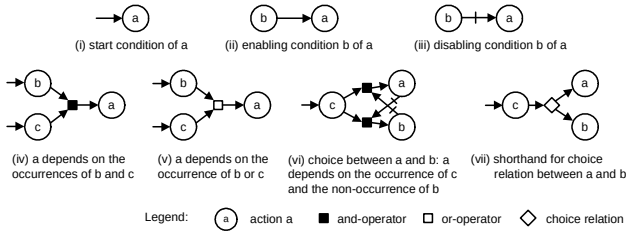


Figure 7. Causality relations in ISDL

In ISDL a behaviour is represented as a rounded rectangle. Containment of one behaviour by another, such as a provider behaviour containing one or more interfaces and a provider composition behaviour containing multiple provider behaviours, is represented by behaviour instantiation. A *behaviour instantiation* represents that a particular kind of behaviour is created in the context of the behaviour that contains the instantiation. We refer to the created behaviour as a *behaviour instance*. The instantiation identifies the kind of behaviour by its name and assigns an instance name to the created behaviour as well; e.g., in Figure 5, behaviour B2 instantiates behaviour B1, such that an instance of B2 contains an instance of B1, called b. The benefit of using behaviour instantiation in this way is that multiple instances of the same behaviour can be created. The relation between behaviour and behaviour instance is similar as the relation between class and object.

Behaviours in a composite behaviour can be related using: (i) interaction activities that relate the interaction contribution activities of the component behaviours; and/or (ii) entry and exit points that represent a causality condition entering a behaviour or a causality condition exiting a behaviour, respectively. Entry and exit points are represented by triangles that point into or out of a behaviour, respectively. Interaction contributions of a component behaviour can contribute to interactions of their composite behaviour. This is represented by drawing a line between the interaction contributions of the component and interaction contributions of the composite. Figure 8 depicts a composite behaviour in ISDL. It shows two behaviours that are related by interactions. The provider behaviour is a composite of two interface behaviours. These interface behaviours contribute to the interaction contributions of the provider behaviour (represented by the circle segments in gray), and are related by an enabling condition that exits one behaviour and enters the other. Normally, we represent a behaviour and its instantiation separately (so in Figure 8 there would be a behaviour OrderInterface and an instantiation o). However, for brevity, we represent them as one.

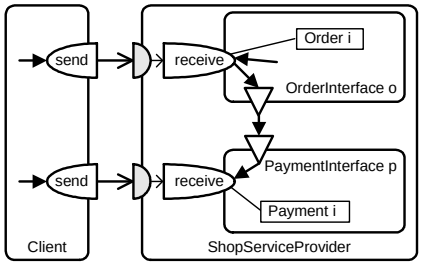


Figure 8. Representation of composite behaviour in ISDL

4.3 Milestone design with the concepts

For design from the perspective of a particular milestone, we often need only a selection of the concepts from Figure 4.

4.3.1 Business process concepts

A business process is a set of related business tasks that are performed to achieve a certain goal. A business process may assign tasks to roles and specify which business partners are authorized to perform which roles. To represent processes, tasks, roles and business partners we use (composite) behaviours, internal activities, behaviours and entities, respectively. Optionally, behaviour instantiation can be used in a business process to represent phases in the execution of the process. Also, information types can be used to represent the structure of information that is established in tasks. Interaction related concepts from Figure 4 are not needed for business process design and neither are the provider composition and interface behaviour concepts. Figure 9 shows an example of a business process in ISDL. This example also shows that entry and

exit points can be parameterized to pass information between behaviours.

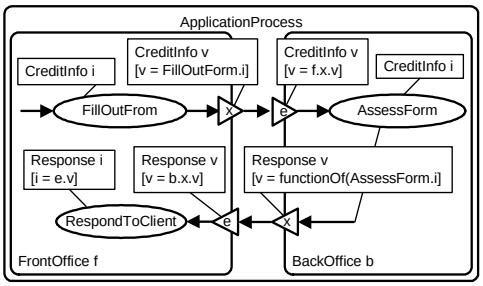


Figure 9. Example business process in ISDL

4.3.2 Application service specification concepts

Current service description techniques describe an application service as a set of related send and receive events that a service provider uses to send messages to and receive messages from a client. Services can be provided at different ports (also called interfaces). We can use the corresponding service-oriented design concepts to represent service specifications. We can use information types to represent the structure of messages that are sent or received. The provider composition behaviour and interaction concepts are not needed for service specification. The ShopServiceProvider behaviour in Figure 8 is an example of a service specification in ISDL.

4.3.3 Application service design concepts

An application service design consist of a composition of services, message exchanges between these services and send and receive events that the composite service makes available to its environment. Hence, an application service design can be represented using the composition behaviour, interaction activity, provider behaviour and send and receive event concepts. We can use information types to represent the structure of messages that are exchanged. Figure 10 shows an example of the internal design of a sales service as a composition of the interacting services provided by a seller and a shipper.

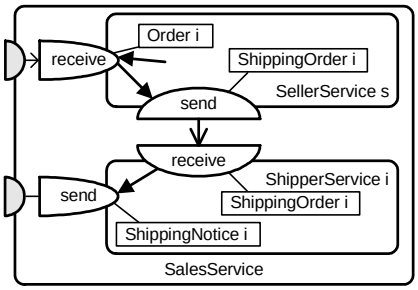


Figure 10. Example service design in ISDL

5. EXAMPLE

This section presents the design of a context-aware "call-a-cab" application, which uses position information to inform a cab about the location of a client, and vice-versa. The aim of this example is to illustrate our service-oriented design approach, the use of the proposed milestones to structure the design process, and the application of ISDL for service modelling.

5.1 Business process

Behaviour CAB_process in Figure 11 defines the "call-a-cab" business process model, representing the tasks that have to be performed, and their relationships. This behaviour definition consists of four behaviour instantiations, which are represented by a behaviour block describing the name of the resulting behaviour instance, its entry, exit and interaction contributions (if any).

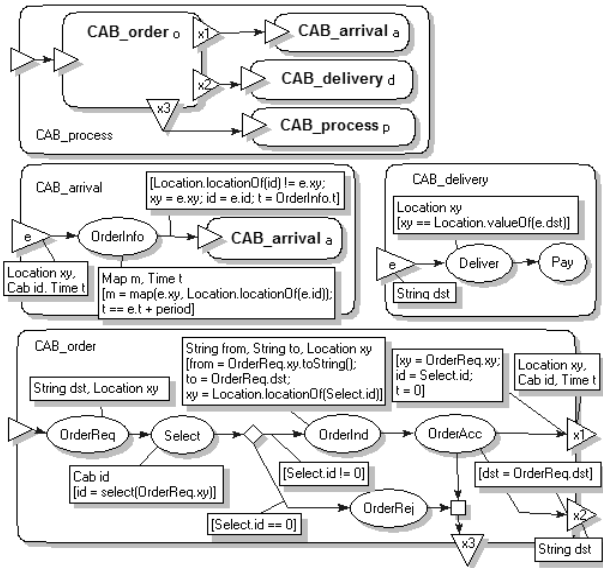


Figure 11. 'Call-a-cab' business process

Behaviour instantiation CAB_order o defines the creation of an instance, called o, of behaviour CAB_order, which handles the ordering of a cab. Action OrderReq models the activity of requesting a cab in which the destination and location of the client are established. The request is followed by the selection of a cab, which is identified by some id, as modelled by action Select. Operation selectOf() represents the algorithm used to determine a (free) cab in the vicinity of the client. An id value of 0 represents no cab is available. Action OrderInd models that the driver is informed about the new order, in case a cab is selected. Subsequently, the client is informed that the order has been accepted, which is modelled by action OrderAcc. Observe that in ISDL, attribute constraints may also be linked to causality relations. In case no cab is available, the order is rejected. Action OrderRej models the notification of this to the client.

Both in case of an accept and a reject, behaviour CAB_process is instantiated recursively, modelling the handling of a new cab request. Only in case of an accept, an instance of behaviour CAB_deliver and an instance of the recursive behaviour CAB_arrival are created. CAB_deliver models the delivery of the client to the destination and the payment. CAB_arrival enables the client to monitor the arrival of the cab. Action OrderInfo models the presentation of a map to the client, showing the current location of herself and the cab. This action is repeated every period time units, until the cab has arrived. Behaviour CAB_deliver and CAB_arrival are made independent, since the core task of delivering the client should not depend on the nice feature of showing the arrival of the cab.

5.2 Application service

This milestone defines which tasks of the "call-a-cab" business process require application support, and which do not. We assume that all tasks as modelled by CAB_order and CAB_arrival have to be supported by a single application. Furthermore, we decide that clients and cab drivers are the application users, i.e., form the environment of the application. This implies that each task is considered as an interaction activity between the application and one of its users, except for the task of selecting a cab, which is considered an internal activity. The tasks modelled by CAB_deliver are considered interactions between the client and the cab driver, and therefore internal to the application's environment.

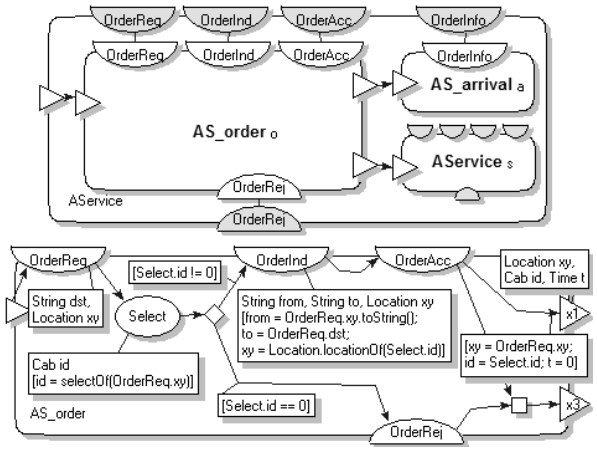


Figure 12. 'Call-a-cab' application service

Behaviour AService in Figure 12 represents the application service. The application service defines the contribution of the application to each of the interaction activities identified above. The constraints on these contributions are defined by behaviours AS_order and AS_arrival. The definition of these behaviours is similar to the ones of the previous milestones, except with actions being replaced by interaction contributions (except for action Select). In this way, the application is made responsible for implementing all constraints on the actions identified in the business process. For brevity, only behaviour AS_order is shown.

5.3 Application service design

This milestone produces an initial design of the application service. We assume that the application functionality is distributed over the mobile phones of the client and cab driver, and a central server, which are connected via a mobile network. Figure 13 depicts the application service design (ADesign) modelled as the composition of the services provided by the application entities on the mobile phones of the client and cab driver (AClient and ADriver, resp.), and the application server entity (AServer). Interactions ClientReq, ClientRsp, ClientInfo, DriverInd and DriverRsp have been introduced to model the interaction between the application entities. For brevity, action attributes have been omitted.

Conformance assessment

In order to assess the conformance of the application design to the application service, we use the second technique mentioned in section 3.2. This means we have to abstract from the design information that has been added in this milestone, and subsequently compare the obtained abstraction to the application service. To illustrate this process, the following simplifications are made: we consider a single client, ignore action attributes and assume a client is informed only once about the arrival of the cab (no recursion). Figure 14 depicts the resulting behaviours of the application service and the application design.

The design information added in this milestone consists of interactions ClientReq, ClientRsp, ClientInfo, DriverInd and DriverRsp. A method has been defined for ISDL to abstract from these interactions [24]. The first step in this method consists of replacing the interactions by actions, which must integrate all constraints defined by the contributions of the interactions. The next step consists of abstracting from, i.e. removing, these actions, which are called *inserted actions* (in grey), since they have been inserted during the refinement steps towards this milestone. The other actions are called *reference actions*, since they provide the reference points in the application service and design for assessing conformance. To perform the abstraction, rules have been defined which obey the following conformance criteria:

1. an indirect relation between reference actions defined via an inserted action in the application design must be replaced by an equivalent relation defined directly between the corresponding reference actions in the application service;
2. similarly, indirect relations between attributes should be replaced by direct relations.

In case of this example, it is straightforward to see that when following these rules, the obtained abstraction of the application design is equivalent (even identical) to the application service.

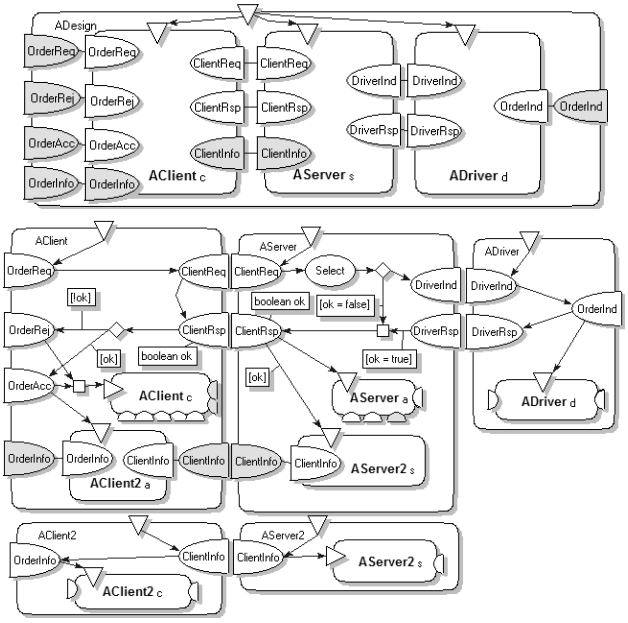


Figure 13. Application service design

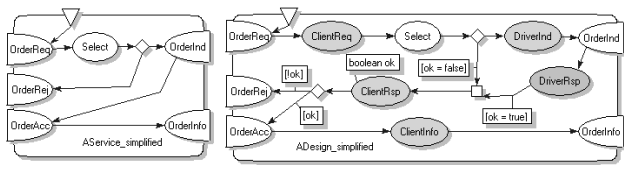


Figure 14. Conformance assessment

5.4 Refined application service design

This milestone decomposes the application server into:

- a *locater*, which allows one to request the geographical location of a mobile phone;
- a *selector*, which selects a cab and asks the driver for confirmation. Selection involves obtaining cab location information, determining availability and choosing the cab closest to the client;
- a *map provider*, which provides a picture of a route map showing the position of the client and the arriving cab;
- an *updater*, which updates the client with aforementioned route maps. For this purpose cab location information is obtained; and
- a *controller*, which coordinates the handling of a cab request, and the updating of arrival information, using the services provided by aforementioned entities.

Figure 15 depicts the services provided by the entities identified above. For brevity, the behaviour defining the composition of the services, similar to Figure 13, has been omitted.

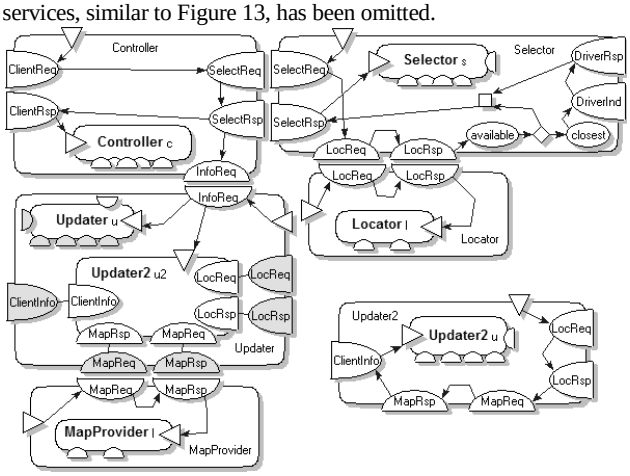


Figure 15. Application server design

Conformance assessment

The conformance of the application server design to the application server service can be assessed analogously to section 5.3. It is left to the reader to show that they do not conform, because interaction contribution ClientInfo may happen independently of interaction contribution ClientRsp. This can be solved by making interaction contribution InfoReq dependent on contribution ClientRsp in behaviour Controller. Although this example is rather simple, our conformance assessment method can be applied to any refined ISDL behaviour.

5.5 Application service implementation

Figure 16 gives an overview of the specifications and designs made so far and their conformance relations.

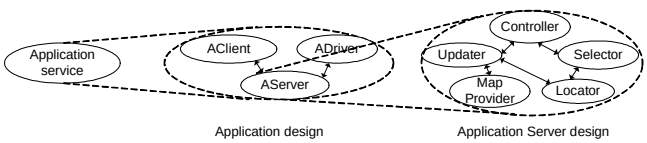


Figure 16. Overview of the design process

The design process ends when services can be provided by available software components or their implementation requires no further design steps. For example, the Locator and MapProvider services in Figure 15 are provided by web-services of the WASP platform [15] on top of which we are currently implementing the 'call-a-cab' application. For this we assume that abstract interactions between application entities are refined into generic communication activities such as one-way and two-way message passing. Furthermore, we work on the development of a tool to transform such communication activities as specified in ISDL, onto BPEL language elements, such as 'invoke', 'receive' and 'reply'. This allows for the generation of skeleton-code, thereby facilitating the implementation of service designs.

6. CONCLUSIONS

This paper identifies service-oriented design as the process of designing an application service such that it can be implemented using service-oriented computing technology. Service-oriented design is needed when the mapping of business process tasks onto available application services is complicated and can not be obtained using predefined decomposition rules. As such, it complements existing techniques for on-line, automated service composition (e.g., [21], [19]), which often assume a close correspondence between business tasks and available application services. Furthermore, service-oriented design advocates the use of platform independent modelling of services.

A systematic and generic service-oriented design approach is presented, characterized by considering recursively the external and internal perspective of an application (part). Design milestones are identified and methods for conformance assessment are described. In addition, a conceptual model for service-oriented design is defined, providing abstract and generic concepts supporting the modelling of business processes, application services and their designs. These concepts have been inspired by existing service description and composition techniques, in order to facilitate their mapping onto the more concrete concepts supported by service-oriented computing platforms. The suitability of ISDL to express the service-oriented design concepts is shown.

Service-oriented design originated from the area of component-based design (for an overview see e.g. [13]). It elaborates on this area by incorporating the principle of distinguishing between externally observable behaviour and internal realization of that behaviour and the principle of integrating applications with business processes. Although these principles are not new, they have special status in service-oriented design methods, our method reflects that. Various research groups have proposed languages for service-oriented design [9], [10], [11], [18]. [18] also supports a form of conformance verification. Our work extends this work, because we describe the role of a modelling language in the design process in more detail and because we consider modelling at higher levels of abstraction. Our work complements the work on design processes for service-oriented design [4], [12], because we take a more precise (formal) approach to modelling and conformance verification. Finally, design languages have been proposed to graphically represent (XML-based) service descriptions (see e.g. [6], [17]). Our work contributes to this area, because we also consider higher abstraction levels. We refer to [14] for a more detailed overview of related work.

We propose ISDL as a language for service-oriented design. From the beginning of its development, we have concentrated on the definition of the design concepts underlying ISDL, aiming at a limited set of generic and elementary concepts. Based on these concepts, a method for assessing the conformance between services and their designs has been defined, thereby providing full support for the service-oriented design approach presented in this paper. Recently, our focus has shifted to the definition of a graphical notation to express the concepts and the development of tool support. An editor is now available and a simulator is almost ready [23]. Tools have and are being developed to partially automate conformance assessment. Furthermore, we work on tools transforming platform independent service designs in ISDL into platform dependent service descriptions, in particular WSDL and BPEL specifications.

7. ACKNOWLEDGEMENTS

This work is part of the Freeband A-MUSE project. Freeband (http://www.freeband.nl) is sponsored by the Dutch government under contract BSIK 03025.

8. REFERENCES

[1] W. van der Aalst. Don't go with the flow: Web services composition standards exposed. *IEEE Intelligent Systems*, 18, Jan/Feb. 2003.

[2] W. van der Aalst, et al. Workflow patterns. *Distributed and Parallel Databases*, 14(3):5–51, July 2003.

[3] J.P.A. Almeida, et al. On the notion of abstract platform in MDA development. In *Proc. of the 8th IEEE Intl. Conference on Enterprise Distributed Object Computing (EDOC 2004)*, Monterey, California, USA, Sept 2004.

[4] G. Alonso, et al. *Web Services: Concepts, Architectures and Applications*. Springer, 2003.

[5] BPMI. *Business process modeling language (BPML) version 1.0*. http://www.bpmi.org/bpml-spec.esp, Nov. 2002.

[6] BPMI. *Business process modeling notation (BPMN) 1.0*. http://www.bpmn.org/Documents/BPMN%201-0.pdf, 2004.

[7] BEA Systems, Microsoft, IBM, and SAP. *Business process execution language for web services (BPEL4WS) version 1.1*. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/, May 2003.

[8] B. Benatallah, et al. *Service-Oriented Software System Engineering: Challenges and Practices*, chapter Service Composition: Concepts, Techniques, Tools and Trends (to appear). Idea Group, Inc., 2004.

[9] B. Benattallah, et al. Conceptual modeling of web service conversations. In *Proc. of the 15th Int. Conf. on Advanced Information Systems (CAiSE)*, Klagenfurt, Austria, 2003. Springer.

[10] B. Benatallah, Q. Sheng, and M. Dumas. The Self-Serv environment for web services composition. *IEEE Internet Computing*, 7(1):40–48, Jan/Feb. 2003.

[11] T. Bultan, et al. Conversation specification: A new approach to design and analysis of e-service composition. In *Proc. of the Int. Conf. on the World Wide Web (WWW)*, Budapest, Hungary, May 2003. ACM Press.

[12] C. Bussler. *B2B integration - concepts and architecture*. Springer, 2003.

[13] J. Chessman and J. Daniels. *UML Components: A Simple Process for Specifying Component-based Software*. Addison-Wesley, 2001.

[14] R. Dijkman and M. Dumas. *Service-oriented design: A multi-viewpoint approach*. Technical Report 04-09, Centre for Telematics and Information Technology (CTIT), University of Twente, Enschede, The Netherlands, 2004.

[15] P. Dockhorn Costa, et al. Towards a Services Platform for Mobile Context-Aware Applications. In *Proc. of the 1st Int. Workshop on Ubiquitous Computing (IWUC)*, Porto, Portugal, 2004.

[16] H. Eertink, et al. A business process design language. In *Proc. of the World Congress on Formal Methods*, 1999.

[17] K. Mantell. *From UML to BPEL*. http://www-106.ibm.com/developerworks/webservices/library/ws-uml2bpel/, 2003.

[18] M. Mecella, F. Parisi-Presicce, and B. Pernici. Modeling e-service orchestration through Petri nets. In *Proc. of the 3rd Intl. Workshop on Technologies for E-Services (TES)*, pp. 38–47. Springer Verlag, Sept. 2002.

[19] B. Medjahed, A. Bouguettaya, A. K. Elmagarmid. Composing Web services on the Semantic Web. In *The VLDB Journal*, 12:333-351, 2003.

[20] OMG. *Model driven architecture (MDA)*. Technical Report ormsc/02-07-01, Object Management Group, July 2001.

[21] B. Orriëns, Jian Yang, and M. P. Papazoglou. A Framework for Business Rule Driven Service Composition. In *Service-Oriented Computing – ICSOC 2003*, LCNS 2910, pp. 75-90, Springer 2003.

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

[23] *ISDL home*. http://isdl.ctit.utwente.nl/, n.d.

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

[25] D. Quartel, et al. On the role of basic design concepts in behaviour structuring. *Computer Networks and ISDN Systems*, 29:413–436, 1997.

[26] UDDI. *Universal description, discovery and integration (UDDI) version 3.0*. Technical report, OASIS UDDI Specification TC, 2003. http://uddi.org/pubs/uddi_v3.htm.

[27] C.A. Vissers and L. Logrippo. The importance of the service concept in the design of data communication protocols. In *Proc. of the IFIP WG6.1 5th Int. Conference on Protocol Specification, Testing and Verification V*, pp. 3-17, 1985.

[28] W3C. *Simple object access protocol (SOAP) version 1.1*. http://www.w3.org/TR/2000/NOTE-SOAP-20000508, May 2002.

[29] W3C. *Web services choreography interface (WSCI) version 1.0*. http://www.w3.org/TR/2002/NOTE-wsci-20020808, August 2002.

[30] W3C. *Web services description language (WSDL): Part 1: Core language version 1.2*. http://www.w3.org/TR/2003/ WD-wsdl20-20031110, Nov. 2003.

[31] W3C. *WS choreography model overview*. http://www.w3.org/TR/2004/WD-ws-chor-model-20040324/, March 2004.

[32] P. Wohed, et al. Analysis of web services composition languages: The case of BPEL4WS. In *Proc. of the 22nd Intl. Conf. on Conceptual Modelling (ER)*, Chicago IL, USA, Oct. 2003. Springer.