Stateful SOA-conformant Services as Building Blocks for Interactive Software Systems

Roman Popp and Hermann Kaindl
TU Wien, Institute of Computer Technology, Vienna, Austria
{roman.popp, hermann.kaindl}@tuwien.ac.at

Abstract

Services implemented through information and communication technology need to provide value for customers, with whom they usually have non-trivial interaction. However, user interface and (Web) service specifications are often disconnected. The most widely used Web services are stateless, hence only trivial user interaction with one-step input and output can be embedded in such a service. Remembering the state is a prerequisite for implementing non-trivial user interaction with a service. We present new stateful SOA-conformant services as building blocks for interactive software systems. This new kind of service has a unified high-level protocol both for (non-trivial) user interaction with a machine and for machine-machine communication. Services with the same protocol can substitute each other (also dynamically at runtime), whether they are machine or user services. Using such services as building blocks, interactive software systems can be composed, also recursively. As a matter of fact, from such service specifications (graphical) user interfaces for non-trivial interaction can be automatically generated.

1 Introduction

Both consumers and enterprises have needs for services, and essential parts of such services can be provided by information and communication technology. Currently, this technology is often restricted to Web services and their choreography, which have restrictions, in particular, for implementing interactive systems (as explained below). The Service-oriented Architecture (SOA) encompasses much more elaborate technology. More recently, SOA has also been focused more on the support of business-relevant services [4].

Interactive systems need to be more and more connected, both for interacting with their users and with each other. Connecting software systems can be achieved, e.g., through Web services. They have well-defined signatures specifying how a Web service can be called, but they are stateless. This allows only embedding trivial user interaction in a single step of input and output in a Web service but no multi-step interactions like a usual flight-booking process. While Web service composition is possible with specific languages, manual effort and glue code is necessary to include non-trivial user interaction. The resulting interactive system cannot be provided as a (SOAP-based) Web service for further composition, since Web services are stateless.

Fig. 1 illustrates a simplified example in the domain of travel booking, where a travel agency system uses two flight booking systems, two hotel booking systems, and a recommender system. The flight booking systems allow booking (single) flights with specific airlines. They are connected with each other and result in a composed flight booking round-trip system, which is included in the travel agency system. All of these systems should be directly accessible by human users as well, without any adaptation, so that each of these systems can either be used by another one, or by a human user, without any additional specification. Flight booking, hotel booking and the recommender system are used together by the travel agency system. Each of these services requires more than one step to fulfill the given task, e.g., (single) flight booking involves first selecting departure and destination airports as well as a flight date, in a second step selecting a flight, and in a third one entering the passenger and payment details. Of course, such a system can be built with stateless (Web) services, as usual with some code in a procedural language. The result, however, is no (stateless) service any more, therefore it cannot be used for further composition.

We strive for providing all such systems through (stateful) services, in order to allow for composing an interactive system, e.g., for a travel agency from such building blocks, which is again a stateful service. Since Web services are insufficient for this purpose, we propose a new form of more complex services, UCP:Services.[1] (UCP stands for Unified Communication Platform, where the technology behind

[1] See `http://ucp.ict.tuwien.ac.at` for details.

URI: http://hdl.handle.net/10125/41303
ISBN: 978-0-9981331-0-2

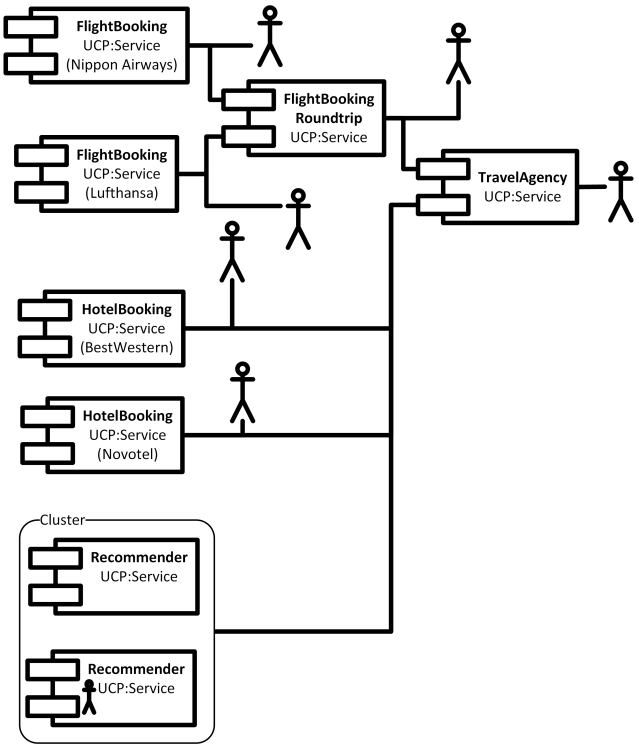

Figure 1: Travel Booking Example

UCP:Services is implemented.) Web services can be used for creating the functionality of UCP:Services, but this is not required.

UCP:Services are stateful, so they can embed non-trivial user interaction. In particular, this interaction is supported by automatically generated (graphical) user interfaces (UIs). For example, FlightBooking can be provided as a UCP:Service, which includes both searching for a flight and (possibly) booking it. This involves more than the trivial user interaction possible in Web services, which only have signatures as their interface specification. In addition to signatures, defined protocols of UCP:Services include especially behavior specifications as their interfaces, which specify such a non-trivial user interaction in the form of behavioral models of possible 'conversations'. The property 'stateful' does *not* mean, however, that a UCP:Service would keep any hidden information from one such conversation to another one.

This kind of protocol also facilitates composition of UCP:Services. In our running example, FlightBooking Roundtrip is composed of FlightBooking (twice), and it is again a UCP:Service. So, it is also directly accessible by a human user (through an automatically generated UI), and it can be further used as a building block for (recursive) composition of the TravelAgency system. The latter is, again, a UCP:Service, which can again be used for further composition.

UCP:Services with exactly the same protocol can substitute each other, whether they are fully implemented in software or just wrap a user interface. The Recommender service in our example can either be an automated recommender UCP:Service or a human providing recommendations through a UI with the same protocol. They may even substitute each other dynamically at runtime, without any technical difference for the composed TravelAgency system.

Our approach for providing UCP:Services builds on Discourse-based Communication Models, which have been originally introduced for specifying communicative interaction models as input for automated generation of (graphical) UIs, see, e.g., [5, 16, 18, 19, 20].

Let us present this work using Design Science Research according to [15], which includes six steps. Our introduction above covers steps one (problem identification and motivation) and two (definition of the objectives for a solution). Step three (design and development) is covered by our presentation of new UCP:Services and their essential properties (in particular SOA-conformance) as well as the definition of our proposed runtime interface of UCP:Services. The presentation of our reference architecture and a brief explanation of its implementation corresponds to step four (demonstration). Step five is an evaluation according to [14]. Finally, step 6 (communication) is given by this paper as a whole.

2 Background and Related Work

First, we present some background material and discuss related work, in order to make this paper self-contained. In particular, we provide some background related to services and to Discourse-based Communication Models, which serve as a technological basis of our new approach.

2.1 Background on SOA

Service-oriented Architecture (SOA) is an architectural style that supports service-orientation according to a set of principles for designing and developing software in the form of inter-operable services. Such services are software components of this architecture, with well-defined functionality and interfaces to be used by other services.

Specific services (according to SOA) provide, e.g., the capabilities of booking a flight or a hotel room. A key idea of SOA is that services can be composed to new services. An example of such a composed service may provide the capability to book a travel, using services for booking flights and hotel rooms, respectively.

Another key idea of SOA is that services can be replaced by other ones during runtime. If the services to be used for

replacement are still unknown, they may be searched for.

Such searches by the requester of a capability are performed according to SOA with a so-called Service Discovery. It uses a (possibly global) Service Registry, where each service registers its provided capability together with its address. If a requester is looking for a specific capability, it asks the Service Registry for an address of a service providing this capability.

The OASIS Reference Model for SOA [11] consists of two models, an Information Model and a Behavior Model. The Information Model specifies the information that may be exchanged with a given service. This includes the format of the exchanged information, the structural relationships, and the definition of used terms. The Behavior Model specifies the possible actions of a given service and the temporal order of single interactions.

An OASIS Service Description provides the information needed to use the described service. It consists of the service interface, the reachability of a service, and possibly some policies (specifications of service performance or quality of service). The service interface has to provide both an Information Model and a Behavior Model.

2.2 Related Work

Web services can be composed, of course. While SOAP-based Web services are stateless, RESTful Web services[2] can have states represented in their resources. This allows defining application state, but not transitions among the states. A comparable approach for handling states, through defining a *session* as a resource or service, is used in Open Grid Services Architecture and Open Grid Services Infrastructure [6].

In order to allow specifying constraints on such transitions, Rauf and Porres [21] extend the WADL[3] definition of RESTful services with pre- and post-conditions. Such conditions can also be defined for SOAP-based Web services using the Semantic Markup for Web Services (OWL-S).[4] In general, however, a process with these states cannot be uniquely specified in this way.

All these approaches are underspecified, as they do not fully define the interaction with the service. Certain approaches using *agents* address this problem by dynamically generating a suitable process (e.g., [23]), based on a semantic service specification. Such a planning task may require a huge calculation effort and result in different processes in different situations. While this may be useful, it makes it hard to define user interaction based on such processes.

Hence, all these approaches cannot define interaction design for user interfaces like UCP:Services, since they lack a complete high-level protocol like UCP:Services. While substitutability is possible between Web services implemented by software, it is not with services provided by humans.

While simple HTML Web pages are stateless, in principle, Web Applications, which use a Web browser as a client, are stateful. This allows them to hold a dialog state and to cache values. For Web Applications, AJAX has established itself as a standard for their interaction with the server. So, interactive applications can be created with this technology. However, no composition of Web Applications (like with Web services) is possible and, therefore, they cannot be used as building blocks for interactive software systems like UCP:Services.

For the direct use of Web services through a human via a (generated) user interface, several research contributions exist. Dynvoker[5] [24] is a modular servlet application, which allows the dynamic generation of GUIs for Web services. However, it can only handle trivial user interaction with one-step input and output through such a generated GUI. For combining the access of independent services, MashUp UIs can be constructed, but cannot be reused by other services / UIs [3].

For specifying interactive software systems based on Web services, MARIAE [13] uses CTT (Concur Task Tree) models. MARIAE focuses on the UI generation for such a system, but the system as a whole cannot be made available as a composed service. Vice versa, since such a model does not define a protocol, only single Web services can be replaced by a human, but not a non-trivial application as whole.

2.3 Discourse-based Communication Models

A simplified flight booking example as shown in Fig. 2 illustrates a Discourse Model for finding and booking single flights as well as their payment. This is an example of one of the three types of models contained in a Discourse-based Communication Model, which we propose for specifying UCP:Services.

In order to keep this explanation brief, let us focus on the excerpt marked with an asterisk, which specifies the selection of departure and destination airports. Both are modeled as an *Adjacency Pair* [10] each (shown as diamonds), connecting each a ClosedQuestion *Communicative Act* and the corresponding Answer Communicative Act (shown as rounded boxes). An Adjacency Pair models typical turn-takings during a conversation (e.g., Request–Response or Question–Answer. Communicative Acts are an abstraction from *Speech Acts*, introduced by [22] and also used

[2] https://docs.oracle.com/javaee/6/tutorial/doc/gijqy.html

[3] https://www.w3.org/Submission/wadl/

[4] https://www.w3.org/Submission/OWL-S/

[5] http://dynvoker.org/

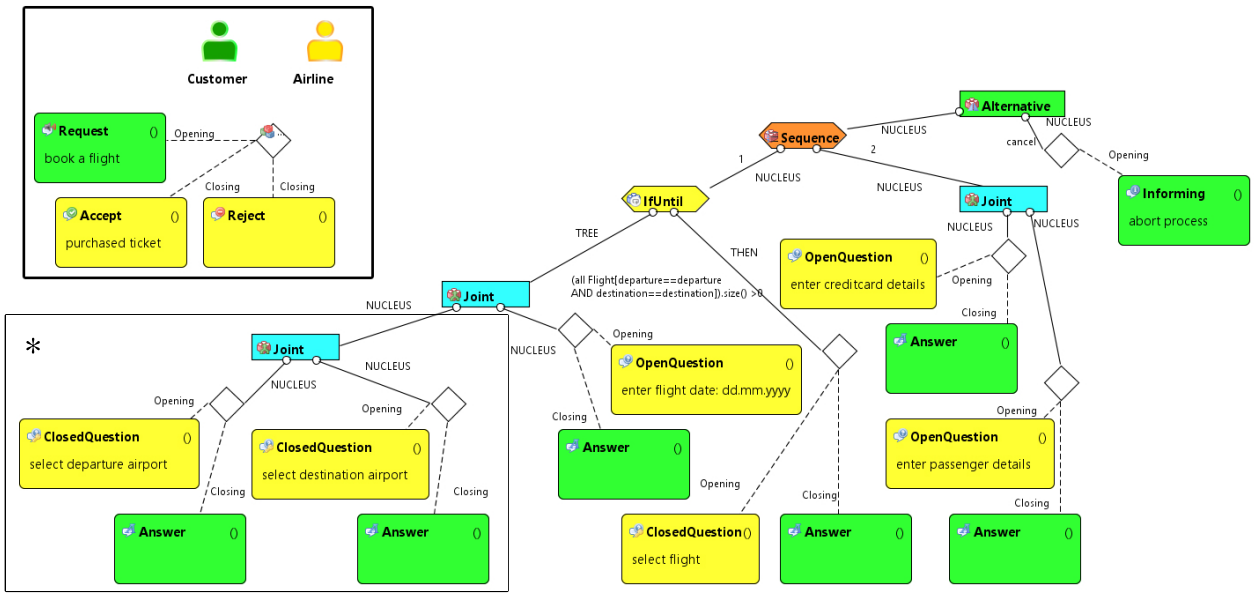

Figure 2: FlightBooking Discourse

as a basis for common agent communication languages, e.g., FIPA ACL [8]. The fill color of the Communicative Acts illustrates the uttering communication party (yellow for the Airline and green for the Customer). The *Propositional Content* of the Communicative Acts specifies the requested functionality of the receiving communication party. In Fig. 2, only a short-hand version is shown. The complete content specification of the 'select departure airport' Communicative Act can be seen in Fig. 5 below. Adjacency Pairs are connected with a *Discourse Relation*, partly derived from Rhetorical Structural Theory (RST) [12], in this example a so-called Joint relation.

Another concept that we adopted from Conversation Analysis is the *Inserted Sequence*. In our approach, it is an additional discourse that a communication party can start in case it does not have enough information to respond to a Request or Question directly. It can interrupt the normal execution of an Adjacency Pair. After the communication party having initiated the Inserted Sequence has all required information, the normal execution of the interrupted Adjacency Pair is continued. In our running example, the main part of the Discourse Model shown in Fig. 2 is actually an Inserted Sequence of the Adjacency Pair in the box in the top left corner.

The other two models contained in a Discourse-based Communication Model are the Domain-of-Discourse (DoD) Model and the Action-Notification Model (ANM). Fig. 3 shows an example of a DoD Model, which fits the Discourse Model example in Fig. 2. In essence, it specifies those objects (and their relations) in the domain that the discourse 'talks' about. DoD Models are represented in Ecore (similarly to UML class diagrams). The Action-Notification Model specifies the actions and notifications that can be executed by a communication party, i.e., its functional interface. Fig. 4 shows an example of an ANM. In fact, this is an excerpt of the predefined Actions and Notifications in UCP. If needed, a developer has the possibility to model additional Actions and Notifications in the ANM.

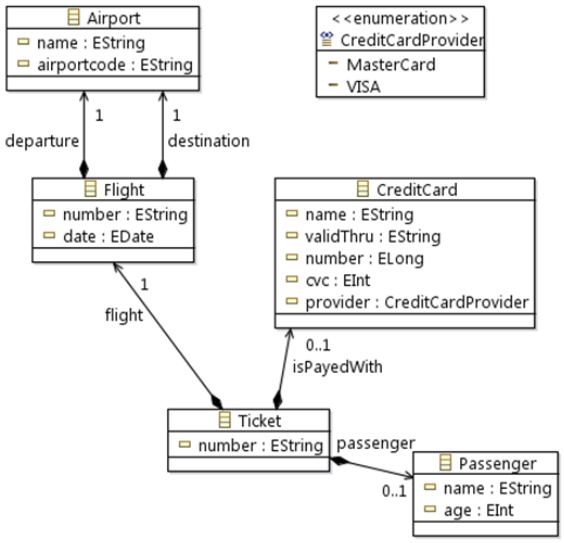

Figure 3: FlightBooking Domain-of-Discourse Model

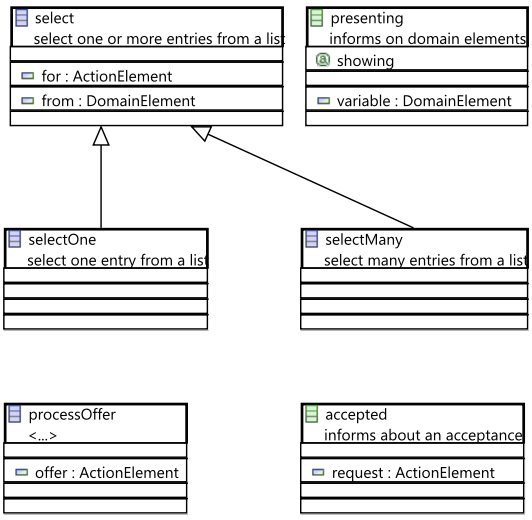

Figure 4: Basic Action-Notifcation Model

3 UCP:Services

Now let us present our new UCP:Services and their essential properties. We define these services and sketch their conformance to SOA. In particular, we explain how UCP:Services can be composed, also recursively, so that they can serve as building blocks for interactive systems.

3.1 Definition of a UCP:Service

A UCP:Service encapsulates a software component that may have a state. The interaction with the service is specified through an interface definition, including a static 'message-signature' part and a dynamic part that defines its behavior. The static part only defines the method for receiving messages. For the dynamic part, Discourse-based Communication Models are used.

Each Discourse-based Communication Model can be automatically transformed to a protocol statemachine, which defines the communication between two UCP:Services. This protocol statemachine can be further transformed to a statemachine from the viewpoint of each actor, for driving its interaction and calling the assigned functionality. These transformations have been previously defined in the context of automated GUI generation [16, 19].

The functionality of a UCP:Service can be provided by a human through such a GUI (UCP:UserService), or by a software system (UCP:MachineService). In general, a UCP:UserService can be an interactive system that includes software functionality as well, which can be provided by UCP:MachineServices it is composed of. A UCP:MachineService is completely automated through software, where (basic) functionality may be provided by usual Web services. In general, a UCP:MachineService may include hardware functionality as well, e.g., movement of a robot.

3.2 SOA Conformance

For sketching the SOA conformance of UCP:Services, we make use of the OASIS Reference Model. We show that everything required by this reference model is available in UCP:Services. In particular, we show this for the two models of OASIS and the encompassing Service Description.

- Information Model

  All the information according to the OASIS Information Model is captured in the DoD Model of UCP. In particular, the DoD Model includes the format of the exchanged information, the structural relationships, and the definition of used terms, in an object-oriented way.

- Behavior Model

  Regarding to the OASIS Reference Model, we have to have a closer look and to consider its two sub-models, the Action Model and the Process Model.

  The Action Model characterizes the actions that can be invoked in a service, or at least the public view of the actions. We use our Action-Notification Model for specifying this for UCP:Services.

  The Process Model deals with the temporal order of single interactions in the sense of messages from and to a service. Our Discourse Model fully specifies it for UCP:Services. Hence, no glue code is needed in this regard, in contrast to Web services.

  More precisely, SOA requires that a service invocation is triggered by the service requester. In order to achieve this with a UCP:Service, its Discourse Model has to ensure that the Requester starts the defined conversation. Technically, this can be modeled as a Request Communicative Act triggering an Inserted Sequence as shown in Fig. 2 above.

- Service Description

  So, the specification of a UCP:Service in the form of a Discourse-based Communication Model provides models according to an OASIS Service Description. However, these models neither specify how an invocation of a UCP:Service can actually be performed at runtime, nor how the messages implementing a conversation are exchanged. This is defined for UCP:Services in their Runtime Interface, which is presented below. In addition, also the technical reachability of a UCP:Service is not defined in these models. So, a unique identification of each

UCP:Service is needed as well as the address of the hosting server (including the transport protocol used). Each UCP:Service has a unique service name, and the address has to be known by the Requester, possibly taken from a Service Registry. Finally, UCP:Services currently do not implement policies, but this is not mandatory for SOA-conformance.

3.3 How to Compose UCP:Services

UCP:Services can be composed from other UCP:Services. Since UCP:Services are stateful and more complex than Web services, also the composition of UCP:Services is more complex than the composition of Web services. So, we have to explain specifically how this can be done.

In order to keep this explanation simple, let us do it with our running example, more specifically the composition of the UCP:Service FlightBookingRoundTrip using two UCP:Services FlightBooking as shown in Fig. 1 above. The interaction with FlightBooking (see Fig. 2) asks for information on the departure airport, the destination airport, and the departure date. FlightBookingRoundTrip asks additionally for information on the date for the inbound flight. It also proposes a second list of flights for selecting an inbound flight. Such an enhanced Discourse Model has to be provided for the composed service in order to specify its protocol.

The more intricate task is to compose the logic of the composed service by using the protocols of the ones it is composed of. Let us explain it as a scenario as it unfolds when the composed service works according to its protocol while using the other ones. After FlightBookingRoundTrip has been invoked, it requests all possible departure and destination airports from both FlightBooking UCP:Services, and merges them. Now it can enter the protocol with its requester and display these merged lists. After the requester has selected both airports and given the destination and arrival dates, the composed service forwards all this flight information to the FlightBooking services and asks for all available outbound and inbound flights. More precisely, it has to swap the airport information for defining the inbound flight, of course. It merges the respective flight lists and presents them to its requester. Analogously, booking and paying flights are composed.

Since such a composition of UCP:Services results in a UCP:Service, the composed one can be used for further composition, i.e., *recursive* composition is possible.

3.4 Unified Usage

Each UCP:Service can either be used by a machine or a human, without any change or annotation. A GUI for human use can even be automatically generated, see [19, 20].

3.5 Substitutability

UCP:Services with the same protocol can substitute each other (also dynamically at runtime), whether they are UCP:MachineServices or UCP:UserServices. Technically, only the address of the used UCP:Service has to be changed, or alternatively, the implementation of the other service has to be bound to the address of the substituted one. The first possibility is easy to achieve through a UCP:Service Registry. This registry allows for finding all the UCP:Services with the same protocol (specified through the same Discourse-based Communication Model), which may substitute each other. It may even automatically deliver different UCP:Services depending on the time of the day. This makes sense for a service like the Recommender for a travel agency, where humans may provide this service during working hours, and a machine otherwise.

4 Runtime Interface

Now let us define the new runtime interface for serial communicative interaction between two communication partners each. These can be a user interface (representing a human user) and a (software) system, or two (software) systems.

At runtime, only messages have to be exchanged. As motivated above, however, their dynamic coordination should be specified explicitly in behavioral models, which can be understood as protocol-machines. We use Discourse-based Communication Models for such specifications. They can either be directly interpreted by both communication parties at runtime or transformed to statemachines before, so that they can be executed more efficiently. While Discourse-based Communication Models, in principle, model all possible behaviors from a view from the top, their interpretation or transformation has to be done from the point of view of the system whose interaction they drive. Automatic generation of such statemachines is possible according to [16], where also subsequent automatic generation of behavioral models for GUIs is defined (for handling the cases where on one GUI screen more than one interaction can be offered at the same time).

For allowing several conversations at the same time, each one is assigned a unique ID. Such an ID is contained in each message, so that it can be related to the conversation it belongs to. For making the approach technology-independent, the messages are encoded in XML for being exchanged. Their content is derived from the DoD Model, which also defines the serialization.

The runtime interface also uses Communicative Acts for the transport of messages between the two interacting com-

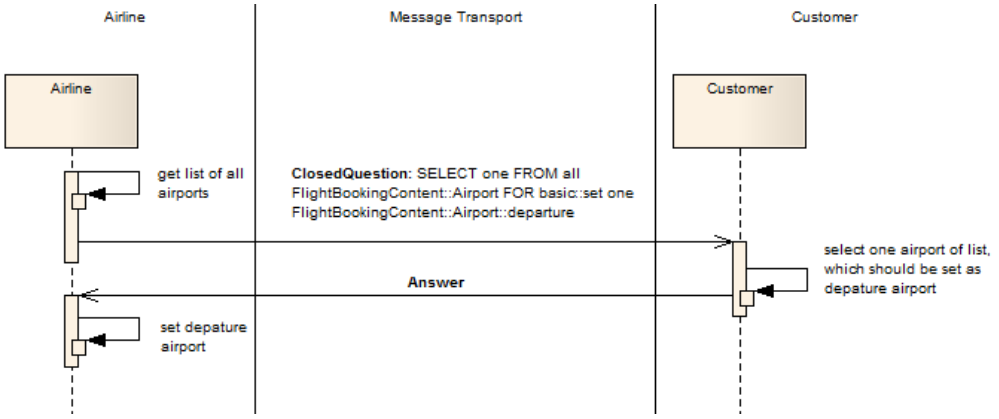

Figure 5: Runtime Message Flow (based on [18])

munication parties. Fig. 5 shows how the communicative interaction unfolds for the simple question and answer as specified in the part of Fig. 2 indicated with an asterisk.

In this example, it is more precisely a ClosedQuestion (for all available airports). Therefore, it is necessary to create the list of possible answers (the list of available airports), before it can be sent to the other system. Once this list is available, the message is encoded in XML and actually sent. The communication partner has to select one of these choices and to send it back in a new message, which is again encoded in XML. The system receiving this reply message can act accordingly.

For acting as described above, several functions have to be available to these systems, e.g., for creation of the choice list, for selecting from it, and for acting based on the selection. They are explained below, and more details are available in [18]. The key point is that it does not matter whether these functions are provided by software or a human (through a user interface).

5 Reference Architecture and Implementation

For facilitating the use of our approach, we created a reference architecture and provide an implementation.[6] Our proposed runtime environment as implemented in UCP is called UCP:RT.

5.1 Overview

Fig. 6 illustrates the structure of our reference architecture. It is based on the structure of common application servers (such as JBOSS,[7] etc.). The main component of this architecture is the UCP:Server hosting the UCP:Services and the UCP:TransportStacks (shown in a yellow box each). A UCP:Service represents the interactive system, with the exception of the message exchange. A UCP:TransportStack is responsible for exchanging the messages. It is possible to implement different kinds of UCP:TransportStacks. In the context of composing systems provided by different servers, the HTTP UCP:TransportStack is used. This stack uses an HTTP/SOAP-based Web service[8] for exchanging the XML-encoded messages. If all systems are hosted on the same host, it is also possible to use a different stack and direct method invocation, which is more performant, of course.

To handle several conversations as sessions at the same time, the concept of a dialog represents them. So, in addition to the three components explained above, there is also a class Dialog (shown in a gray box in the figure). Each Dialog instance has a relation to exactly one UCP:Service and one UCP:TransportStack instance each, hosted on the same instance of the UCP:Server. Each UCP:Service provides a separate ServiceDialogPort, used for receiving Communicative Acts for each ongoing conversation. For each conversation, UCP:RT provides an instance of the Dialog class for each communication party. Therefore, a conversation between two UCP:Services consists of two instances of the Dialog class, which are connected through UCP:TransportStack instances. This is illustrated with the dashed line in Fig. 6.

5.2 Reusable Components

The components of our reference architecture can be reused for any interactive system providing such a runtime interface. The UCP:Server and each UCP:TransportStack are independent of a specific UCP:Service, which is the only component that depends on the specific system.

[6] Eclipse update site at `http://ucp.ict.tuwien.ac.at/eclipse-update`

[7] `http://www.jboss.org/`

[8] Web Service available at `http://ucp.ict.tuwien.ac.at:18888/HttpServer`

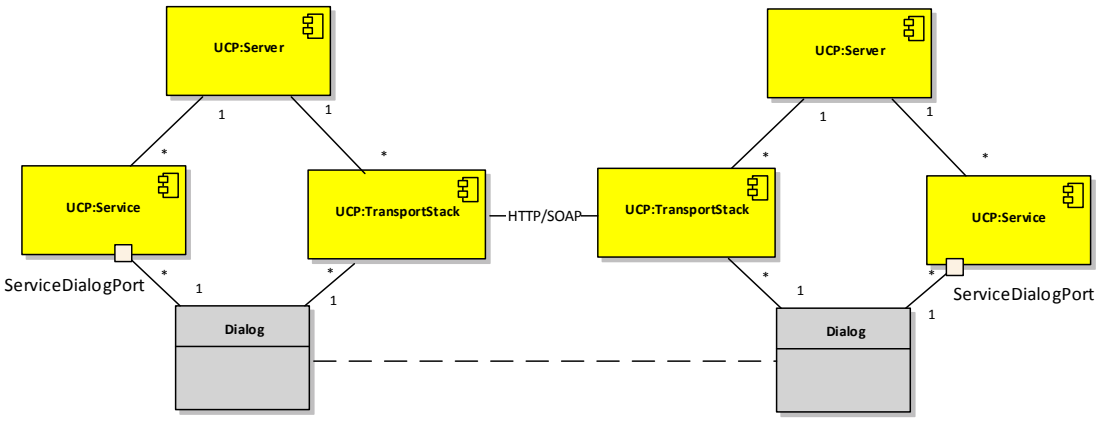

Figure 6: Structure of the Runtime Architecture

A UCP:Service represents a communication party. It can either be implemented as a (software) system or as a user interface, which allows a human user to interact with another communication party.

Each UCP:Service provides two interfaces. The first is called ServiceDialogPort and provides the functionality to receive Communicative Acts. To facilitate simple session management, for each dialog/conversation a new instance of the ServiceDialogPort is created. The second interface of the UCP:Service is mainly used to request a new instance of a ServiceDialogPort as defined in the *Factory Pattern* [9].

5.3 GUI/Software-specific Components

A UCP:Service can either be provided by a human through a user interface or by a software system. Therefore, we provide two corresponding reference designs for UCP:Services. For a user interface, the reference design follows the MVC pattern. It is explained in [17].

For a software system, two main components exist: one for interpreting the Discourse-based Communication Model or the statemachine transformed from it, respectively (the *execution engine*), and one for providing the specific application functionality of the software system. The first component is independent from a specific model or can be generated fully automatically, and can, therefore, be reused for each system.

So, only the application functionality has to be implemented/provided specifically. What functionality is needed can be extracted from the Discourse-based Communication Model and is based on the Action-Notification Model and the Domain-of-Discourse Model, and it is independent from the behavior model (with the restriction that all preconditions for a specific functionality have to be fulfilled, e.g., for selecting a flight, a departure and a destination airport have to be set). If such a functionality is available through Web services, these can be directly connected to the provider of the application functionality.

The current implementation of the *execution engine* directly interprets the Discourse-based Communication Model, but also an implementation as a statemachine (which is fully automatically generated) can be provided easily.

5.4 Implemented Examples of UCP:Services

The example introduced above is fully implemented on top of UCP:RT, including the (generated) GUIs.[9] Our unified interfaces provide substitutability of user services with software services, or vice versa. For example, we used (an earlier version of) this unified interface both for connecting (generated) GUIs and for interfacing robots with other robots [7].

In effect, this allows for a novel form of service composition, as illustrated in the example of Fig. 1 and explained above. All these (composed) services are implemented with our reference architecture and its implementation, and can be accessed through the runtime interface, see above. Service discovery is already possible with such UCP:Services. The corresponding Service Registry is implemented as a UCP:Service as well [19].

In addition, our approach and its reference implementation allow a service to be used by different GUIs. For example, we have tailored GUIs for different devices for the FlightBooking service [20].

[9]The (generated) GUI for the FlightBooking service can be found at `http://ucp.ict.tuwien.ac.at/UI/FlightBooking`, and the GUI for the TravelBooking service, which uses the FlightBooking service, at `http://ucp.ict.tuwien.ac.at/UI/travelBooking`.

Table 1: Comparison of Approaches

	Approach	API Definition	Stateful Process Definition	Stateless Recursive Composability	Stateful Recursive Composability
Web service based approaches	Web services	x		x	
	Web services with pre- and postconditions	x	o	x	
UI Frameworks	MariaE		x		
	Mashup UIs		x		
	Web-Apps		x		
	UCP-Services	x	x	x	x

6 Evaluation

Evaluation according to Design Science Research in [14] distinguishes several evaluation method types. We provide an *illustrative scenario* in the Introduction section, which is also implemented in a *prototype* using our new UCP:Services as explained above. This prototype enables the automatic generation of GUIs for each UCP:Service and their execution, as well as the composition of new UCP:Services using existing ones (even recursively). GUIs can be generated automatically, and in the course of generation tailored for different devices (smartphones, tablet computers and desktop PCs) [20].[10] In effect, each change of some service can quickly and cheaply result in newly generated GUIs.

As a *logical argument*, we compare all the competing approaches simply based on their key properties. The result of this comparison can be seen in Table 1. An "x" means that the approach in the given line has the property of the given column. An "o" means that a property is partially fulfilled. Let us explain the properties used for this comparison:

- *API Definition* means, as usual, the availability of a machine-readable interface definition.
- *Stateful Process Definition* means a definition of a process with states, for non-trivial interaction with a given service.
- *Stateless Recursive Composability* means that services can be composed, also recursively, but none of the (composed) services has a Stateful Process Definition.
- *Stateful Recursive Composability* means that services can be composed, also recursively, and they can have a Stateful Process Definition.

Web services as well as the approaches based on them focus on an API Definition and on Stateless Recursive Composability. The latter can be achieved through *service composition*, e.g., implemented in BPEL [1]. However, this do not provide processes with states for non-trivial interaction. Web services with additional information as well as WADL provide pre- and post-conditions in various forms. From these a planner implemented in an *agent* (see e.g., [23]) could possibly generate processes, but this does not always result in a uniquely defined process. Manual definition of processes is possible, e.g., through BPMN [2]. Such a *choreography* can result in a process with states for non-trivial interaction. It cannot, however, be packaged as a service again, and thus not recursive composition is possible.

The UI framework approaches, in contrast, focus on the interaction with a human user. In order to facilitate non-trivial interactions, they have a Stateful Process Definition. However, they do not support any composition.

UCP:Services (our new approach) fulfill all the listed properties. So, they allow both recursive composition and non-trivial interaction.

7 Conclusion and Future Work

UCP:Services are stateful and more complex than the wide-spread Web services. Hence, they allow also non-trivial user interaction with a service (even through a generated user interface). The protocol for specifying this interaction, a Discourse-based Communication Model, is actually the same as for interaction with a UCP:MachineService. In effect, this is a unified high-level protocol both for (non-trivial) user interaction with a machine and for machine-machine communication. UCP:Services with the same protocol can also substitute each other (even dynamically at runtime), whether they are machine or user services. Using UCP:Services as building blocks, interactive software systems can be composed, also recursively.

In addition, this paper presents a flexible runtime architecture for interactive (software) systems represented as services. It is specified for the runtime environment of user interfaces called UCP:RT. According to this specifica-

[10] see e.g., `http://ucp.ict.tuwien.ac.at/UI/FlightBooking` for our running example and `http://ucp.ict.tuwien.ac.at/UI/accomodationBooking` for more real-world GUIs

tion, UCP:RT is implemented in the Unified Communication Platform. This runtime architecture allows both user interfaces and services to be run on the same platform, based on Communicative Acts.

The unification of UCP:Services for human users and machine services has the potential to close the gap between the service community and the user interface community. Even more importantly, UCP:Services may improve implementations of services for providing value for customers.

In future work, a composition language for UCP:Services should be developed, which is still lacking. In addition, our approach needs to be evaluated in a larger real-world context.

References

[1] Web services business process execution language version 2.0. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html, April 2007. [Online; accessed 23-August-2016].

[2] Business Process Model and Notation (BPMN). http://www.omg.org/spec/BPMN/2.0, Jan. 2011. [Online; accessed 08-February-2015].

[3] F. Daniel, S. Soi, S. Tranquillini, F. Casati, C. Heng, and L. Yan. Distributed orchestration of user interfaces. *Information Systems*, 37(6):539 – 556, 2012. {BPM} 2010.

[4] T. Erl, P. Chelliah, C. Gee, J. Kress, B. Maier, H. Normann, L. Shuster, B. Trops, C. Utschig, P. Wik, and T. Winterberg. *Next Generation SOA: A Concise Introduction to Service Technology & Service-Orientation*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1st edition, 2014.

[5] J. Falb, H. Kaindl, H. Horacek, C. Bogdan, R. Popp, and E. Arnautovic. A discourse model for interaction design based on theories of human communication. In *Extended Abstracts on Human Factors in Computing Systems (CHI '06)*, pages 754–759. ACM Press: New York, NY, 2006.

[6] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. The physiology of the grid. *Grid computing: making the global infrastructure a reality*, pages 217–249, 2003.

[7] H. Kaindl, R. Popp, D. Raneburger, D. Ertl, J. Falb, A. Szep, and C. Bogdan. Robot-supported cooperative work: A shared-shopping scenario. In *Proceedings of the 44th Annual Hawaii International Conference on System Sciences (HICSS-44)*. IEEE Computer Society Press, January 2011.

[8] Y. Labrou. Standardizing agent communication. In J. G. Carbonell and J. Siekmann, editors, *Multi-agents systems and applications*, pages 74–97. Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[9] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (2nd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2002.

[10] P. Luff, D. Frohlich, and N. Gilbert. *Computers and Conversation*. Academic Press, London, UK, January 1990.

[11] C. M. MacKenzie, K. Laskey, F. McCabe, P. F. Brown, and R. Metz. Reference model for service oriented architecture 1.0. http://docs.oasis-open.org/soa-rm/v1.0/, 2006.

[12] W. C. Mann and S. Thompson. Rhetorical Structure Theory: Toward a functional theory of text organization. *Text*, 8(3):243–281, 1988.

[13] F. Patern, C. Santoro, and L. D. Spano. Engineering the authoring of usable service front ends. *Journal of Systems and Software*, 84(10):1806 – 1822, 2011.

[14] K. Peffers, M. Rothenberger, T. Tuunanen, and R. Vaezi. *Design Science Research in Information Systems. Advances in Theory and Practice: 7th International Conference, DESRIST 2012, Las Vegas, NV, USA, May 14-15, 2012. Proceedings*, chapter Design Science Research Evaluation, pages 398–410. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[15] K. Peffers, T. Tuunanen, M. Rothenberger, and S. Chatterjee. A design science research methodology for information systems research. *J. Manage. Inf. Syst.*, 24(3):45–77, Dec. 2007.

[16] R. Popp, J. Falb, E. Arnautovic, H. Kaindl, S. Kavaldjian, D. Ertl, H. Horacek, and C. Bogdan. Automatic generation of the behavior of a user interface from a high-level discourse model. In *Proceedings of the 42nd Annual Hawaii International Conference on System Sciences (HICSS-42)*, Piscataway, NJ, USA, 2009. IEEE Computer Society Press.

[17] R. Popp, H. Kaindl, and D. Raneburger. Connecting interaction models and application logic for model-driven generation of Web-based graphical user interfaces. In *Proceedings of the 20th Asia-Pacific Software Engineering Conference (APSEC 2013)*, 2013.

[18] R. Popp and D. Raneburger. A High-Level Agent Interaction Protocol Based on a Communication Ontology. In C. Huemer, T. Setzer, W. Aalst, J. Mylopoulos, N. M. Sadeh, M. J. Shaw, and C. Szyperski, editors, *E-Commerce and Web Technologies*, volume 85 of *Lecture Notes in Business Information Processing*, pages 233–245. Springer Berlin Heidelberg, 2011. 10.1007/978-3-642-23014-1_20.

[19] R. Popp, D. Raneburger, and H. Kaindl. Tool support for automated multi-device GUI generation from discourse-based communication models. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '13)*, New York, NY, USA, 2013. ACM.

[20] D. Raneburger, H. Kaindl, and R. Popp. Strategies for automated GUI tailoring for multiple devices. In *Proceedings of the 48th Annual Hawaii International Conference on System Sciences (HICSS-48)*, pages 507–516, Piscataway, NJ, USA, 2015. IEEE Computer Society Press.

[21] I. Rauf and I. Porres. Designing level 3 behavioral restful web service interfaces. *SIGAPP Appl. Comput. Rev.*, 11(3):19–31, Aug. 2011.

[22] J. R. Searle. *Speech Acts: An Essay in the Philosophy of Language*. Cambridge University Press, Cambridge, England, 1969.

[23] W. Shen, Q. Hao, S. Wang, Y. Li, and H. Ghenniwa. An agent-based service-oriented integration architecture for collaborative intelligent manufacturing. *Robotics and Computer-Integrated Manufacturing*, 23(3):315 – 325, 2007. International Manufacturing Leaders Forum 2005: Global Competitive Manufacturing.

[24] J. Spillner, I. Braun, and A. Schill. Flexible human service interfaces. In *Proceedings of the 9th International Conference on Web Services*, pages 79–85, 2007.