Posr: A Comprehensive System for Aggregating and Using Web Services

Mohammed AbuJarour*, Mircea Craculeac◇, Falko Menge◇, Tobias Vogel◇, Jan-Felix Schwarz◇
Hasso-Plattner-Institute for IT Systems Engineering
Potsdam, Germany
* *firstname.lastname@hpi.uni-potsdam.de*
◇ *firstname.lastname@student.hpi.uni-potsdam.de*

***Abstract*—Recently, the number of public Web Services has been constantly increasing. Nevertheless, consuming Web Services as an end-user is not straightforward, because creating a suitable user interface for consuming a Web Service requires much effort. In this work, we introduce a novel approach where user interface fragments for consuming Web Services are generated automatically, and aggregated and customized by end-users to match their preferences. Users can collaboratively improve the auto-generated user interfaces and share them among each other. Our three main sources of Web Services are explicit registration, automatic identification and collecting over the Web, as well as extraction and generation from existing web applications. We validated our approach by implementing it as a comprehensive system coined *"Posr"*.**

***Keywords*-Web Service; Service Repository; Composite; UI Fragments;**

I. Getting the Most of SOA

Service-Oriented Architecture (SOA) has changed the way services are delivered in private (closed) environments, such as organizations, as well as public environments, such as the WWW. The number of *public* Web Services offered over the Web has been increasing all the time. For example, the Service-Finder project [1], was able to find more than 27,000 Web Services over the Web.

This wealth of Web Services is utilized by aggregating and combining multiple Web Services together to achieve new or user-defined tasks. As user-defined tasks can be created arbitrarily, the challenge is to create a proper user interface (UI) for consuming such a composite service. In this work, we propose a novel approach to deal with this challenge and several associated issues.

A. Example: The EU Services Directive

One particular example for the service markets emerging is the European Union (EU). Until 28^{th} of December 2009 all EU Member States have to implement the "Directive 2006/123/EC on services in the internal market" [2], which has been introduced by the European Parliament and Council and aims to create a genuine internal market for services. Therefore, the directive seeks to remove legal and administrative barriers in order to make it easier for businesses to set up in other Member States of the EU and to provide services cross-borders or on a temporary basis.

The Internet is going to play an important role in the implementation of this directive. On the one hand, it will be the marketplace for trading services across country borders. On the other hand, governmental organizations will have to open their services to be used over the Web. Furthermore, private sector service providers may add value to these Web-based public sector services by combining them with other services, which may also be of use when performing certain formalities.

One example for such a value-chain containing public and private sector service providers has already evolved in the United Kingdom (UK). In particular, the UK limited company has become quite popular across the European Union, because it is a company structure with limited liability but without the need to pay up capital. Recently, third-party service providers emerged, which handle all formalities of registering a new company with the authorities and offer a broad range of additional services, e.g., a bank account, a registered office, or a company secretary.

The current implementations for this example contain either lots of media breaks, if the government services are used directly, or are hand wired compositions with custom-developed software by third-party providers. Throughout this paper, it is shown how these services can be extracted (wrapped), discovered, and composed, and how a media-break-free user interface can be created.

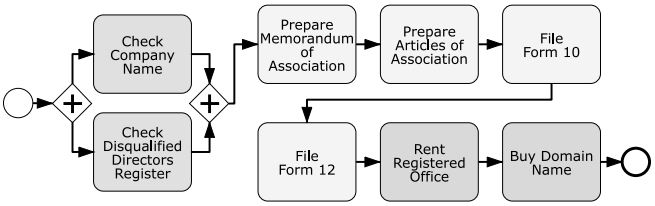


Figure 1: Example process of a value-added service for the formation of a UK limited company.

Figure 1 shows a process model using the Business Process Management Notation (BPMN) [3] of a value-added service for the formation of a private company limited by shares in the United Kingdom. The first six activities of the process are services of the UK Companies House [4], which are required for every new limited company. In the

978-0-7695-3708-5/09 $25.00 © 2009 IEEE
DOI 10.1109/SERVICES-I.2009.111

beginning, it has to be checked whether the desired company name is not already in use and the directors are not disqualified from leading a company. The according registries can already be queried online through the website of the UK Companies House. Section III-A shows how these Web forms can be transformed into SOAP-based Web Services. For the remaining steps, electronic forms are provided in the Portable Document Format (PDF). We implemented these activities as Web Services manually, to show examples illustrating the challenges of user interface generation and editing in Section III-C. The last two activities in the model are Web Services by private sector service providers, which have been discovered on the Web as shown in Section III-B. In Section III-D, the process model depicted in Figure 1 serves as a blueprint for aggregating a media-break-free user interface for the entire service composition.

B. Contributions of this Work

The main contributions of this work are:

- Extraction and generation of Web Services to wrap existing web applications.
- Automatic identification and gathering of Web Services over the Web.
- Automatic generation of user interface fragments that are able to consume Web Services.
- Aggregation of user interface fragments for service compositions.

The remainder of this paper is organized like this: a general overview of our approach is given in Section II. Further details about each component of the system are introduced in Section III. In Section IV, we point to our online system. We close with a conclusion and outlook in Section V.

II. Overview

A composite application, such as the formation of a UK limited company, requires an appropriate user interface to collect inputs from the user and pass them to the service. The structure of a user interface to consume a Web Service is highly-coupled with the data structures being exchanged during the consumption of the Web Service. This causes much effort during the creation of UIs for consuming applications, which effectively limits reusability of such Web Services, whereas reusability is one of the main advantages of the SOA. This contradiction motivates the automatic generation of user interfaces for consuming Web Services.

Constructing valuable composite services needs a rich collection of Web Services, where vast alternatives are available. To meet this requirement, we rely on a service registry and repository component, which we call the *"Depot"*. This component manages all considered Web Services and provides lookup and retrieval features for Web Services. An important feature of our service *Depot* is the automatic identification and gathering of public Web Services over the Web. To increase flexibility, feeding individual Web Services into the *Depot* is also possible. A special case of such feeds is the outcome of the *"Faster"* component, which is used to extract and generate Web Services from web applications (forms). For more details about *Faster*, please, refer to Subsection III-A.

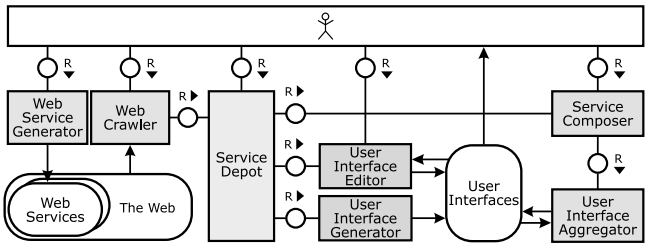


Figure 2: Constitutes of *Posr* System (FMC Block Diagram).

The architecture of our system, *Posr*, is depicted in Figure 2, which illustrates its main constituents and the interactions between them. Application scenarios are split into two sets: The first set is the discovery and provision of Web Services, and the second set includes composition and consumption of Web Services.

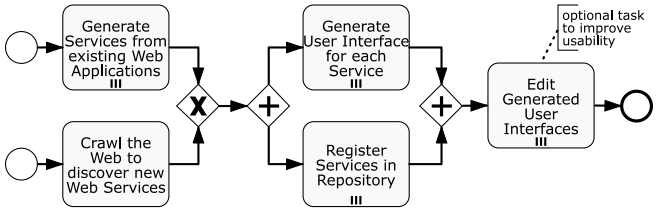


Figure 3: BPMN process model showing the discovery and provision of services.

Figure 3 depicts the scenario of discovery and provision of Web Services. It has two starting points: an extracted Web Service from a web application using *Faster* or a URL where public Web Services are published. In the latter case, the *Depot* crawls the provided URL and gathers public Web Services published there. Registering a Web Service in the *Depot* involves automatic generation of UI fragments for each operation in the Web Service. This scenario ends with an optional step where the end-user can customize the automatically generated UI fragments to match his preferences and increase their reusability.

The second scenario is depicted in Figure 4 and starts with a search for suitable Web Services in the *Depot*, which returns a list of service operations associated with their pregenerated UI fragments. Optionally, these UI fragments can be customized to suit individual needs. After that, the user could either choose to consume the individual Web Services separately or combine them to a composite application. The latter case involves aggregating UI fragments to make up a new UI for the composite service, which can then be consumed by the user.

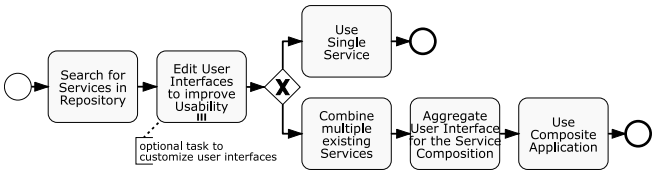


Figure 4: BPMN process model showing the composition and consumption of services.

III. POSR: LOCATE AND AGGREGATE WEB SERVICES

After introducing the main constituents of *Posr* in Figure 2, this section details each of them and explains how they interact with each other to complete the entire mission.

A. Faster: Providing Web Services for Web Applications

The Web provides an huge number of valuable and interesting services in form of web applications using HTML forms. Nevertheless, the vast majority of these applications is intended to be used solely by humans, but not by software programs, i.e., they are *human-oriented services*. While the number of websites offering Web Service interfaces is gradually increasing, there are still many websites, which do not offer such interfaces. Thus, their functionality remains inaccessible for computer programs. Unveiling this functionality through Web Service interfaces allows to use it in service compositions and mashups.

The goal of this component is the automatic extraction and generation of wrappers for plain, human-oriented applications offered on web pages, which are then accessible via Web Service APIs. For brevity, we will refer to this as *creating Web Services*, which in turn is necessary to realize the vision of Semantic Web Services [5].

1) Deriving Web Application Wrappers from HTML Forms: HTML forms are the common way of collecting user's input data on websites or – more precisely – for the applications offered on those websites. In contrast to plain text content, HTML forms follow a rather systematic style. Therefore, HTML forms are the ideal starting point to "bridge the gap" between human- and machine-accessible applications. Some applications require the user to fill in several forms on consecutive web pages. These forms usually differ in number and kind of the form elements, they contain. It is common that some form elements *occur multiple times*, e.g., to correct or confirm previous inputs. Multiple occurrences of the same form elements share the same semantic and usually the same value and therefore represent the same piece of information.

The challenge is to wrap and expose all these forms and their input elements in a way that the underlying application is accessible in a single service invocation. Therefore, the generator has to build an internal model including the occurring forms, their form elements and the sequence of the web pages, on which they occur. To build up this model, the entire process that a user normally passes through has to be analyzed. Such an analysis produces much better results if a human-assisted approach [6] is used.

In our approach called *Faster* we monitor user inputs and track them over the entire process. In each step, form elements are analyzed and matched against the current model. If they cannot be matched, they are treated as new form elements and examined regarding their descriptive text labels and whether they are writable. Afterwards, they are added to the model as well.

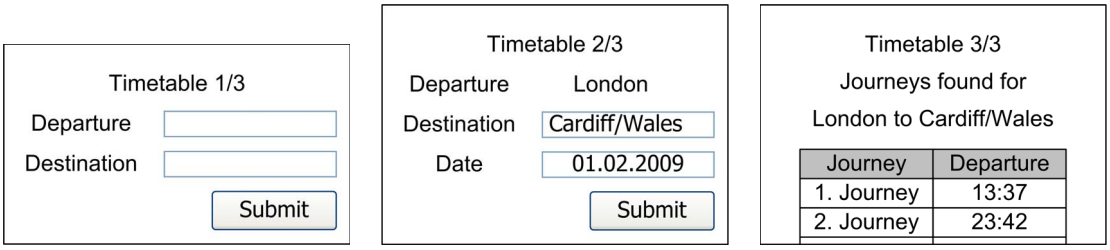
Thorough investigation of such web pages revealed that they are often not created manually, but rather using automatic software tools. We observed that corresponding form elements *usually* have the same internal names, i.e., attribute names. If this is not the case, more or less fuzzy hints for correspondences have to be considered for the matching, e.g., surrounding descriptions [7] or `<label>` elements.

As form elements can be associated with each other, a chain of corresponding form elements is manifested in a so-called model element, as illustrated in the example in Subsection III-A2. These model elements are also contained in the internal model of the generator. When a Web Service is generated, the internal model is used to derive inputs of the created Web Service. The generator also defines mandatory, optional and unspecifiable model elements for the actual service invocations. Only mandatory and optional model elements become input parameters of the Web Services. Besides form elements, also meta-data, such as labels and length limitations, are recorded in order to annotate the input parameters.

The results that such web applications produce can have a wide range of data types, generally classified into simple or complex types. The class of simple types includes strings, numbers, URIs, etc, whereas, the class of complex types include files or structured fragments of data, e.g., tabular or object information. Simple types include strings, numbers, or URIs, In most cases, the result of the entire web application service is presented in HTML on the final page. Therefore, the output of the created Web Service is defined by an XPath expression that extracts one fragment of the HTML document in case of simple types or multiple HTML segments, in case of complex types. We implemented an algorithm, which parses a common type of simple tables into lists of objects, whose attributes are inferred from the table header.

2) Faster by Example: Journey Planner: In this scenario, the user needs to find an appropriate journey from London to Cardiff on the 1^{st} of February. On the first page (Figure 5a), departure and destination names are requested. Both fields are *new* to the internal model and filled by the user with "London" and "Cardiff", respectively. These fields are added to a list of input parameters and thus, become mandatory query parameters for the Web Service to be generated.

On the second page (Figure 5b), both values are presented again. However, the departure has already been recognized



(a) Entry page querying for departure and destination.

(b) Second page having recognized the station "London", corrected the station "Cardiff" and presenting the date chooser.

(c) Result page presenting the entered information and the journeys found according the query.

Figure 5: Possible process of using a timetable website.

and thus, is not editable at this stage. Instead, it is shown as a static string. The destination value has been filled with an automatically suggested value by the web application to meet the correct station name. It is included in the form again to enable the user to change it, if necessary. The form element, whose value is "Cardiff/Wales", has the same internal name as the departure field on the first page, i.e., it is recognized as *reoccurring*. Therefore, it is not added again to the list of input parameters, but the new value is noticed. Additionally, a new field was added to collect the desired date of the journey. It is pre-filled automatically with the current date. Hence, this field becomes an optional input parameter for the generated Web Service.

On the third page shown in Figure 5c, no further forms can be found and so the execution stops here. The result is a table with available journeys.

This table can be parsed into a list of journey objects containing the journey description and the journey time with the help of an XPath expression. The internal model of the generator after tracking the user's input is depicted in Figure 6.

Model Element	Form 1	Form 2	Interface
A	A		A (man)
B	B	B	B (man)
C		C	C (opt)

A = Departure
B = Destination
C = Date

X = new X = reoccurring

Figure 6: Illustration of the internal model of the generator at the end of the monitoring process with mandatory (A and B) and optional (C) input parameters.

For the registration of a UK limited company as described in Subsection I-A, the Companies House website [4] can be used to get information about availability and present ownership of company names by entering the company name or number in an HTML form. Furthermore, certain filter criteria can be specified. After passing through the generation process, the company name and number input fields have been recognized as mandatory parameters. The filter checkboxes became optional parameters, since even leaving the filter unchanged yields a valid result. There is also a hidden input field, which results in a static, unspecifiable parameter. Similar to the journey planner, the result is a table, which can be parsed with an XPath expression.

B. The Depot

The *Depot* represents the heart of *Posr*, because all Web Services, their associated UI fragments and composite Web Services are managed by this component. The maintained wealth of Web Services in the *Depot* comes from end users who wish to incorporate their favorite Web Services in the *Posr* to benefit from its novel features. The remainder of this Subsection details this constituent and explains its role.

Starting with the direct association between *Faster* (Section III-A) and the *Depot*, the user employs *Faster* to extract and generate a Web Service from a web form. The extracted Web Service is then registered automatically in the *Depot*. Similarly, the user can trigger a registration request for a single Web Service to the *Depot*. Registering a Web Service in the *Depot*, regardless of its source, involves generating UI fragments for each operation in the registered Web Service. For further details on generating these UI fragments, please, refer to Subsection III-C.

The *Depot* can automatically identify Web Services over the Web, collect them, and register them.

1) Automatic Identification and Gathering of Web Services Over the Web: Our approach of automatic identification and gathering of Web Services over the Web employs web crawling techniques. In [8], the authors introduce several crawling strategies and types. In this stage, we focus on SOAP-based Web Services only, which means that it is not necessary to visit a URL more than once; thus, a snapshot crawling strategy fits our needs. This kind of strategy visits each URL only once; rediscovered URLs are rejected as duplicates. This strategy is augmented with focused crawling techniques, which limit the crawling scope by gathering only websites that conform to a predefined set of deciding rules.

All in all, a focused crawling type combined with a snapshot strategy fits our needs.

To identify a SOAP-based Web Service, we consider the following three criteria and configure our crawler accordingly. First, we focus on links ending with the term "wsdl", which indicates that the link *may* point to a WSDL file. Second, we issue a request to the link and investigate the *MIME* type in its response. The MIME type should be one of these types: *"text/xml"*, *"application/xml"*, or *"application/wsdl+xml"*, because a valid WSDL file should have one of these content-types. Finally, the document should contain at least the WSDL and SOAP namespaces. The aforementioned three criteria govern our crawler and help identify WSDL files on a specified URL, but do not guarantee that the found documents are valid WSDL files.

To overcome this limitation, a WSDL validation step is deployed by the *Depot* whenever a Web Service registration request is triggered.

2) The Depot as an Integration Environment: All constituents of *Posr* communicate with the *Depot* to perform their tasks. Some communication channels are not visible to the end user, e.g., the interaction between the UI generation component and the *Depot*, whereas other communication channels are visible, e.g., the interaction between the *Depot* and the service composer part. In this part, we explain each type of communication between system constituents and the *Depot*.

The *Depot* has an implicit communication channel with any service provider on the Web. This communication channel allows the *Depot* to identify public Web Services over the Web, gather and validate them, and register valid services in the system. The *Depot* maintains a similar, but explicit, communication channel with *Faster* to register the automatically extracted Web Services from existing web applications in the *Depot*.

Registering a Web Service in the *Depot* involves issuing a request to the UI generation component to generate UI fragments for all operations in the Web Service. The generated UI fragments can be customized by the user by rearranging components or changing their layout for easier and more convenient consumption of the service. The updated UI fragments are also stored and managed by the *Depot*. Users can share their UI fragments and collaborate on creating or reusing UI components.

Composing Web Services is done through the service composition component, which relies on the *Depot* to provide all the necessary information, e.g., the URLs of the Web Services, and UI fragments of their operations. The composite service and its aggregated UI is stored back in the *Depot* and managed as a new service.

C. Generating and Editing Graphical User Interfaces for Web Services

One of the main claimed advantages of *SOA* is service composition and reuse. Service compositions can be modeled using a business process diagram and executed via an orchestration engine. However, the graphical user interfaces of consuming applications have to be implemented manually. If the effort of building a client application and adapting it to the existing services is almost as high as creating a client application and new services from scratch, the latter option is likely to be chosen and existing services are not reused anymore.

In this Subsection, we introduce an approach to create user interfaces for single operations of Web Services using as much automatic generation as possible. These user interfaces serve as building blocks for user interfaces of service compositions.

1) Services with Complex Data Structures: Creating an appropriate UI to consume a Web Service has to reflect all data types exchanged by the service. This task incorporates many challenges if the service operates on complex data types, which require domain experts to create the appropriate UI for consuming them, e.g., services provided by large enterprise applications may expect to receive input documents consisting of hundreds of data fields. In such cases it is even hard for service providers to come up with a user-friendly graphical user interface. Not only should service consumers supply services with correct data types, but also the output of a service's response may be hard to interpret.

To overcome this drawback, user interfaces could be designed collaboratively if a service provider is not willing to provide an appropriate one. This goal is achieved in the best way in an open, web-based environment where folksonomy effects could lead to publicly available UIs of high quality, as experiences of multiple users are combined to interpret complex data structures.

To enable reusability and multi-modality of UIs, we need a UI specification language that separates data, logic, and presentation layers. As we do not assume any previous user knowledge of such a language and Web Service technologies, automatic generation and graphical tooling are necessary to make building UIs for services as easy and convenient as possible.

2) Transforming WSDL documents to XForms and Editing Generated XForms: User interfaces for consuming Web Services can be generated automatically from WSDL descriptions. In this context XForms is the UI specification format of choice. The XForms standard addresses the limitations of classic HTML forms and introduces form descriptions with a clean Model-View-Controller separation using XML mark-up. Moreover, the data collected and processed by an XForm is represented as XML, which enables seamless interoperation between UIs and Web Services. XForms can

directly invoke services using the *submission* element and process service responses.

Due to the match of concepts the transformation of WSDL documents to XForms has been subject of past work and research activities [9], [10]. The main problem is the generation of appropriate form controls based on the backing schema. There are also several tools available that realize XML Schema to XForms transformations [11]–[13], and therefore could be applied as a foundation to build upon.

Evaluating existing approaches revealed that they neither leverage all the information available inside WSDL documents nor use all features of the XForms specification. Furthermore, most of the available XSL transformations are only byproducts of other projects and no longer actively maintained. WSDL2XForms [14] turned out to be the most advanced implementation. Hence, we decided to enhance it with lessons learned from evaluating other approaches in order to utilize it in our application.

A generated form consists of two views: one for gathering service's input data, and the other one to display the response. Labels of form controls are derived from schema element names. If any annotation that documents a schema element is available, it can be used to provide a *hint* for the correlative *input* field.

The UI generation is initiated by the *Depot* when a new service is registered. The generator produces technically fully functional forms for each operation of each port type and stores them in the *Depot*. The forms can be used to test the service, and as a starting point for manual customization.

Automatic generation of UI for Web Service consumption is only limited to technical aspects, but semantic information of services play an important role for the interface design, e.g., predefining some demanded input values, providing hints and constraints on inputs, and convenient arrangement of form controls to make a form usable for end-users. Therefore, the user is able to adjust the generated form to fit usability needs. This is achieved using the *XForms Editor* which enables visual editing of forms and runs directly in a web browser.

The *XForms Editor* is based on the Oryx modeling platform [15]. The editor is a *WYSIWYG (What-You-See-Is-What-You-Get)* tool for creating and adjusting XForms by users, who do not have deep knowledge about the standard. Modified forms are stored back in the *Depot* and can be used and improved by any user, which results in a situation, where several alternative user interfaces are available for each service.

D. Aggregation of User Interfaces for Service Compositions

Having a way to create graphical user interfaces for single service operations is already a big step towards end-user enablement, since it allows for easy testing of services. However, the key idea of SOA is not to use a service just on its own, but rather to create composite applications by combining multiple services into service compositions. This leads to the question whether it is possible to combine the graphical user interfaces in a similar way than the services themselves.

1) Concepts of User Interface Aggregation: Considering the example described in Subsection I-A we could now start combining services of the UK Companies House with some private sector services. As shown in Figure 1, this kind of service composition can be described with a process model, e.g., using the Business Process Modeling Notation (BPMN) [3]. The activities in this kind of process models are the operations of one or more services and the control flow describes the order, in which they are invoked. A novelty in our approach is to model service compositions with the help of a Web-based process modeling tool as shown in [15], rather than employing a fat client application, which would have to be installed on the user's machine. Expecting users of a service composition tool to be experts in business process modeling would be unrealistic. Therefore, only a basic subset of BPMN 1.2 is used, which comprises *Tasks*, *Sequence Flows*, *Exclusive* and *Parallel Gateways*, as well as *Start* and *End Events*. This common subset has been selected based on the study presented in [16]. At this stage automatic or semi-automatic service composition techniques can be applied. However, this is out of scope for our current work. Know, the idea is to leverage the information of the process model to not only generate an executable composite service, but also to aggregate the user interfaces of the involved service operations into a user interface for the composite service.

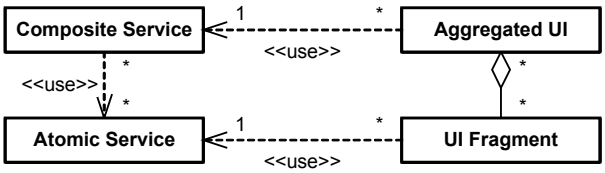


Figure 7: UML diagram showing the artifacts of service composition and user interface aggregation.

To get an overview of what has to be done by such an aggregation one should have a look at the involved artifacts. Figure 7 shows the dependencies among service and user interface artifacts. First of all, a service composition uses multiple atomic services and thus acts as a service consumer to them. The same accounts for UI fragments, since each of them is capable of invoking a single operation of a single atomic service. This means that both the composite service and the according group of UI fragments can be leveraged to access the services involved in a composition. This observation leads to two approaches on how UI aggregations can be performed.

On the one hand, one can aggregate the UIs of all the service operations involved in the process and eventually

do the service composition directly on the user interface level, i.e., performing service orchestration with the elements of the user interface description language. This approach has the advantage of a relatively low overhead, but it also means that the service composition is only usable through the UI and not as a standalone Web Service. Furthermore, this approach requires data transformations between service invocations to be handled within the UI, which may be not very efficient depending on the UI technology used and would also violate the separation of concerns paradigm. On the other hand, the more advanced way is to generate a new user interface for a composite service by combining parts of the operation UIs and reconfigure them to invoke a new service, namely a Web Service created out of the process model orchestrating the services underling the composition. This composite Web Service would also be usable without the graphical user interface and data transformations can be handled with the appropriate technologies. The penalty of using this approach is the need for additional infrastructure, e.g., an orchestration engine.

2) User Interfaces for Service Compositions: The key for being able to perform the aggregations described above is to employ an abstract user interface description language (UIDL). "Abstract" in this case means that it only defines the semantics of a user interface, i.e., the intention behind the UI controls. An abstract UIDL does not restrict the graphical layout, the concrete UI elements employed by the client, and the client technology in general. It was one of the rationales for choosing XForms in Subsection III-C2 that it provides an abstraction level perfectly suitable for tools decomposing and reassembling user interfaces.

In detail, certain patterns are applied when performing the aggregation. First, the XForms *models* of all UI fragments are integrated into the combined UI. XForms *cases* represent the actual user interface screens, through which the user navigates using *triggers*. BPMN *Sequence Flows* are implemented by merging the XForms *cases* from the UI fragments into one *switch* within the aggregated UI. *Exclusive Splits* are decided through the user at runtime or by an XPath query configured at design time in the process model. *Parallel Splits* are realized by nesting the XForms *switches* from the UI fragments into a *switch* that can be controlled by the user. Due to the nature of XForms, it is possible to virtually perform the activities in parallel, e.g., by entering data in one *case* while another one is waiting for the response of its service operation. Since the process of the UK example contains a *Parallel Split* and *Sequence Flows*, the UI fragments of the public and private sector services are combined into an XForms *switch* that acts like a tabbed browsing interface at the beginning and continues as a wizard. This aggregated user interface provides a media-break-free, guided interview experience to the user.

As in the case of generating user interfaces for single operations, the automatically aggregated user interface may not be perfect. However, it is a perfect starting point for further editing with the XForms editor presented in Section III-C2, since the setup for communicating with the underlying services is already done. Finally, the XForms can be executed using different types of clients, e.g., browser plug-ins, JavaScript-based clients, server-side XForms implementations, or native clients for mobile devices. The abstractness of the language also allows for multi-modality, e.g., controlling the user interface using audio output and speech recognition.

IV. Demonstration

We managed to fulfill the above formalisms in a running system, coined *"Posr"*. Figure 8 shows the home page of Posr, which is available at:
http://posr.no-ip.org/

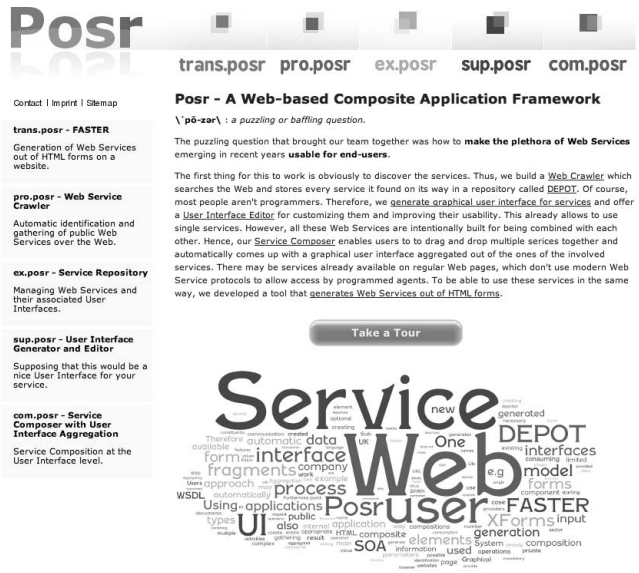


Figure 8: The homepage of Posr.

Posr consists of five components: *trans.posr*, *pro.posr*, *ex.posr*, *sup.posr*, and *com.posr*. *trans.posr* analyzes HTML forms and transposes their hidden functionalities into Web Services. Allocating public Web Services over the Web is the task of *pro.posr* which crawls a specific URL for public SOAP-based Web Services and proposes using the found Web Services by registering them in the system. *ex.posr*, in its turn, manages all discovered Web Services and their associated UI fragments which are pre-generated by *sup.posr*. *sup.posr* supposes that each Web Service will be invoked by the end-user and pre-generates UI fragments for all its methods. Finally, *com.posr* is used to compose Web Services and aggregate their pre-generated UI fragments to provide a unified view for the composite Web Service.

V. Conclusion and Outlook

In this work, we present a novel approach that handles some issues, which hinder end-users using or combining public Web Services. We show how to extractWeb Service-sout of HTML forms by monitoring users interacting with them. We demonstrate gathering public Web Services over the Web using a Web crawler, which employs a focused crawling type combined with a snapshot strategy. A service registry and repository stores all service-related artifacts and allows to search for suitable service operations. We employ automatic generation of user interface fragments for consuming Web Services and enable their Web-based, collaborative editing and sharing. Finally, we introduce an approach to aggregate such user interface fragments for service compositions defined through process models, in order to create entire composite applications. Future work will be to introduce data mappings and reuse between service invocations and to extend the approach to support other types of Web Services, such as RESTful Web Services.

References

[1] "Service Finder," http://www.service-finder.eu. [Online]. Available: www.service-finder.eu

[2] European Parliament and Council, "Directive 2006/123/ec of the european parliament and of the council of 12 december 2006 on services in the internal market," ISSN 1725-2555, pp. 36–68, December 2006. [Online]. Available: http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:32006L0123:DE:NOT

[3] "Business Process Modeling Notation (BPMN) Version 1.2," http://omg.org/spec/BPMN/1.2/, January 2009. [Online]. Available: http://omg.org/spec/BPMN/

[4] "UK Companies House," http://companieshouse.gov.uk/.

[5] S. A. McIlraith, T. C. Son, and H. Zeng, "Semantic web services," *IEEE Intelligent Systems*, vol. 16, no. 2, pp. 46–53, 2001.

[6] S. Raghavan and H. G. Molina, "Crawling the Hidden Web," in *Proceedings of the 27th International Conference on Very Large Databases (VLDB 2001)*, 2001, pp. 129–138.

[7] J. Wang and F. H. Lochovsky, "Data Extraction and Label Assignment for Web Databases," in *WWW '03: Proceedings of the 12th international conference on World Wide Web*. New York, NY, USA: ACM, 2003, pp. 187–196. [Online]. Available: http://dx.doi.org/10.1145/775152.775179

[8] K. Sigurdsson, "Adaptive Revisiting with Heritrix," Master's thesis, Universitiy of Iceland, 2005.

[9] "IBM alphaWorks XML Forms Generator," http://www.alphaworks.ibm.com/tech/xfg.

[10] K. Song and K.-H. Lee, "An Automated Generation of XForms Interfaces for Web Services," in *Web Services, 2007. ICWS 2007. IEEE International Conference on Web Services*, July 2007, pp. 856–863.

[11] "Chiba Project," http://chiba.sourceforge.net/.

[12] "XRX/XForms Generator," http://en.wikibooks.org/wiki/XRX/XForms_Generator.

[13] "xsdTransformer," http://xsdtrans.sf.net/.

[14] "WSDL2XForms," http://wsdl2xforms.sf.net/.

[15] G. Decker, H. Overdick, and M. Weske, "Oryx - An Open Modeling Platform for the BPM Community," in *Proceedings of the 6th Int'l Conference on Business Process Management (BPM 2008)*, ser. LNCS, M. Dumas, M. Reichert, and M. C. Shan, Eds. Milano, Italy: Springer Verlag, 2008, pp. 382–385. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85758-7_29

[16] M. Muehlen and J. Recker, "How much language is enough? theoretical and practical use of the business process modeling notation," in *Proceedings of the 20th Int'l Conference on Advanced Information Systems Engineering (CAiSE 2008)*, Z. Bellahsène and M. Léonard, Eds. Montpellier, France: Springer Verlag, 2008, pp. 465–479. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-69534-9_35