Using Service Workflows and a Declarative UI to enable Collaborative Task driven Applications

Arjen Schoneveld and Frank Lyaruu

Dexels BV, Asterweg 20D2, 1031HN, Amsterdam, the Netherlands
aschoneveld@dexels.com, flyaruu@dexels.com
WWW home page: http://www.dexels.com/

Abstract. The ever increasing number of features in typical (business) front-end applications has become a challenge for our memory capacities. Users have to navigate a plethora of menus, sub-menus and sub-sub-menus to perform a certain task that adheres to the current state of a business process workflow. Service Oriented Computing has enabled flexible composition of business processes in contrast to monolithic application architectures. Still, we mostly use bloated front-end applications that have been created to integrate with those services. In order to achieve a truly agile and pro-active Service Oriented Eco-system, the user interface has to become Service Oriented as well, without compromising the richness and responsiveness of typical desktop user interfaces. We propose a three faceted solution to this problem, consisting of UI annotated Web services, a declarative user interface framework and Service Oriented workflows to enable pushed (mailed), short-lived, miniature front-end applications. In this paper we will introduce each of these facets and describe a use case in the context of a real-life application: a Sports Management Resource Planning system that is used by a large number of sports associations in the Netherlands.

1 Introduction

In a true computing eco-system, a human user interacts and cooperates with *software colleagues*. A software colleague, represented by a (composed) service, can pro-actively assist a user in his/her work as opposed to relying on a classical information pull by this user. Reaching a user can be achieved by actively starting a conversation from a software component, orchestrated by a software workflow, to his human colleague. In technical terms, by pushing specialized mini-applications bootstrapped in an pre-determined initial state. Both the application and the push mechanism can be multi-modal, i.e. various devices and protocols can be involved. This application push is in contrast to normal workflow applications that typically rely on a centralized portal application.

This paper introduces an architecture for creating task driven front-end applications. A business process, modeled by a workflow, pushes tasks to specific users or groups of users. In [6] a particular solution to a specific problem in the context of e-government is described. It describes a system that actively involves the user in business processes using email and workflows to send forms to users. We aim to make a generalization from this particular solution and problem context. Our challenge is to design and implement a software framework that meets a number of requirements. It should be highly configurable, in a sense that not only e-mail can be used to push user tasks, but any other suitable push mechanism such as chat rooms. Furthermore, support for rich user interfaces (mini applications) instead of page based forms, is required. A third requirement is application agility or the ability to gracefully embrace change. A final requirement is that, instead of the *business analyst*, we want to empower the *service developer* with an application framework that can also be used without requiring high level visual tools, leveraging the simplicity of Service Oriented development.

Our solution, that meets these requirements, is based on a complete Service Oriented Computing (SOC) approach consisting of the following main ingredients: a service run-time and programming environment, *Navajo*, and a declarative Service Oriented UI framework and programming language called *Tipi*. The Navajo framework is build around special XML documents, called *Navajo Documents*. A Navajo Document defines a self describing XML service language that contains rich annotated meta-data to support the automatic generation of user interfaces (WSDL annotation approaches are used in e.g. [8] and [9]). A Navajo document contains hints for the user interface about data-type, formatting and data length restrictions, read-only indications and data descriptions. A Navajo Document is the starting point for all Web service implementations. A Web service implementation can be realized by the Navajo Service programming language called *Navascript*. Navascript is all about consuming (request) and producing (response) Navajo Documents while simultaneously performing the service operation.

In turn, the annotated Navajo Documents can be consumed by Tipi to produce rich user interface components with minimal effort. A Tipi based application consists of declarative parts that define user interface components (labels, tables, windows, etc.) that can be bound to Web services. User interface event listeners can be defined declaratively and subsequent procedural steps (actions), to be taken in case of an event, are also defined within a Tipi program.

A Service workflow definition is applied to model the long running aspects of the business process by tracing the path of invoked services using service triggers. Within a Service workflow, the invocation of a service can trigger a state transition in a Finite State Machine. A subsequent state can issue additional service invocations as required by the modeled process. Such service invocations may include pushing applications to a human workflow participant.

In the next section we will introduce the Navajo Service Framework used to implement the Web services and the Service workflows. In the third section Tipi, a declarative service UI framework, will be described. Service workflows and Tiplets, as final ingredients for our Task Driven Applications, will be introduced in section four and five respectively. Section six discusses the feasibility using a use case from a Sports Association business process. Finally, we will present some conclusions.

2 Navajo Service Framework

The Navajo Service Framework introduces a generic service language (1), a service programming language (2) and a service run-time (3). Navajo advocates a specific message oriented approach approach as opposed to an object encapsulation approach that is followed by many popular Web service frameworks like Apache Axis[1]. The benefit of using message- over object-oriented in distributed applications is discussed in large detail in [5]. The Navajo framework supports in many ways the paradigm shift from purely Object Oriented development to Service Oriented Development[13].

Navajo was designed according to the following main design principles: (1) *Embrace change* and (2) *Rapid development and deployment*. The service language, Navajo Document, enables us to annotate data elements with enough meta-data for generating rich user interfaces with minimal amounts of code. In the setting of large scale *SaaS* solutions we may benefit from SOC as a programming paradigm[1]. SOC offers both flexibility and integration requirements that are important for SaaS solutions. Navajo Document follows a single schema approach that enables rapid development of contract first services[7]. The construction of the service contract, specified as a Navajo Document, and the implementation of the actual service are combined in a single document: a Navajo script. Navajo scripts make it possible to create flexible services on the fly. These scripts can be written in Navajo's own format: *Navascript*, or any other scripting language supported by Java, such as JavaScript, JRuby or plain Java. Within a script arbitrary software components can be accessed for performing various tasks. A service response, formulated as a Navajo document, is weaved from the data produced by the invoked components. A similar approach is taken in [4]. The main difference is that Navascript uses a single artifact to define both the interface and the implementation of the Web service.

A Navajo Document is used to define structured and typed data, hence a native Navajo Web service does not need a separate XML Schema definition. A Navajo Document contains data in the form of properties which can be grouped into messages. Properties have types, a unique id and possibly additional meta data like input constraints and descriptions. Several data types are supported, among which a binary data type. A message can contain diverse sub-messages and sub-messages of exactly the same structure in which case it is called an array message. Listing 1.1 shows an example Navajo Document.

Listing 1.1. An example Navajo document containig both a simple- and an array message.

```
<navajo>
   <message name="competition">
      <property name="name" value="Primary League" direction="out" ↩
           type="string"/>
   </message>
   <message name="teammatchschedule" type="array">
```

[1] http://ws.apache.org/axis/

```
        <message name="teammatchschedule">
            <property name="matchname" value="Ajax - Feyenoord" ↩
                direction="out" type="string" length="32"/>
            <property name="date" value="2010-01-01" direction="out" ↩
                type="date"/>
            <property name="starttime" value="20:00" direction="out" ↩
                type="clocktime"/>
        </message>
        <message name="teammatchschedule">
            <property name="matchname" value="PSV - Ajax" direction="↩
                out" type="string" length="32"/>
            <property name="date" value="2010-01-08" direction="out" ↩
                type="date"/>
            <property name="starttime" value="20:30" direction="out" ↩
                type="clocktime"/>
        </message>
    </message>
</navajo>
```

A Navajo Document can be accessed and manipulated by the Navajo Service Run-time following the operations defined in a Navascript document. Property values can be addressed using Navajo Expressions which are akin to XPath expressions. The formal definition for addressing a property is:

```
"["["/"]{0,1}<message name>["@"<index>]{0,1}"/"<message name>]
"@"<index>]{0,1}+"/"<property name>"]"
```

Array messages are addressed using their index. Addressing the value of the property `matchname` contained in the second `teammatchschedule` message works as follows:

```
[/teammatchschedule@1/matchname]
```

Also messages can be addressed for the purpose of being able to iterate over messages.

Navascript is an XML language that can be used to define a 'mapping' between a Navajo Document request and internal Java components and, at the same time, a 'mapping' between internal Java components and a Navajo Document response. Internally, a script is automatically compiled to a Java class on first use. A Navascript based Web service can be used as input for automatically creating a WSDL specification. The generated Java class is responsible for mapping Navajo (array) messages to and from Java Bean (array) fields. A Java bean implements a specific system component, like a database adapter, a domain object, an email operation, etc. Property values are mapped to Java types. Besides all primitive Java types (including String and Date), a binary data-type is also provided which can be used to represent images, videos and arbitrary documents. We are leveraging an event driven SAX parser to implement streaming of binary objects. Streaming enables us to serve arbitrarily large binary objects while still serializing to XML. Within Navascript, Navajo Expressions can be used to formulate full expressions in which both user-defined functions as well as Navajo messages and properties can be addresses (specialized XPath). Navascript resembles BPEL when encapsulating each Java component as as service. However, BPEL offers limited data manipulation, since its main purpose is to orchestrate the interaction among Web services[3]. Listing 1.2 shows an example that uses a database component to execute a SQL query. A result set field is used to loop over all records, invoking a Web service for each record and simultaneously constructing two array messages: `matchschedule` and `pools`. Finally, the constructed array messages are joined to form a new array message `teammatchschedule`. Listing 1.1 could be a response from a Web service called `GetTeamSchedule`, represented by this Navascript example.

Listing 1.2. A Navascript for the GetTeamSchedule webservice.

```
<?xml version="1.0" encoding="UTF-8"?>
<navascript xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ↩
    xsi:noNamespaceSchemaLocation="http://www.navajo.nl/schemas/↩
    navascript.xsd">
    <map.sqlquery datasource="'sportlinkkernel'">
        <!-- Use a generic database component to execute a SQL query ↩
            -->
        <sqlquery.query>
            select poolid
            ,      (select c.competitionkind
                    from competitiontype c
                    where c.competitiontypeid = ↩
                        teamcompetitiontypeseason.competitiontypeid
                    ) as competitionkind
            from teamcompetitiontypeseason
            where
            teamid   = ? and
            seasonid = get_current_season
        </sqlquery.query>
        <sqlquery.addParameter value="[/parameters/teamcode]" />
        <!-- Create placeholder array message match schedule -->
        <message name="matchschedule" type="array" />
        <!-- Create array message pools, looping over resultset, and ↩
            calling poolmatchschedule Web service via navajomap ↩
            component -->
        <message name="pools">
            <map ref="resultSet">
                <map.navajomap>
                    <navajomap.createproperty name="'/parameters/↩
                        poolcode'" value="$../columnValue('poolid')" /↩
                        >
                    <navajomap.callwebservice name="'clubsites/nl/↩
                        poolmatchschedule'" append="'/schedule'" ↩
                        appendTo="'/matchschedule'" />
                    <property name="poolcode" direction="out">
                        <expression value="$../columnValue('poolid')" ↩
                            />
                    </property>
                </map.navajomap>
            </map>
        </message>
        <!-- Join array messages matchschedule en pools on poolcode ↩
            creating new array message teammatchschedule ordered by ↩
            date and start time -->
        <map.joinmessage>
            <joinmessage.join message1="'/matchschedule'" message2="'/↩
                pools'" ignoreSource="true" joinCondition="'poolcode=↩
                poolcode'" suppressProperties="'poolcode'" />
            <message name="teammatchschedule" orderby="'date,starttime↩
                '">
                <map ref="resultMessage" />
            </message>
        </map.joinmessage>
    </map.sqlquery>
```

```
</navascript>
```

The Navajo service run-time offers important aspects to the service habitat. Foremost, the automatic script to Java compilation enables hot-deployment of both fresh- and changed services. Several protocol listeners (end-points) have been implemented to optimally support all four different MEP types: *request/response*, *in-only*, *out-only* and *solicit/reply*. Currently, the Navajo run-time offers support for SOAP, HTTP, XMPP(Jabber) and incoming e-mail end-points. Pluggable authentication and authorization can be used to implement the required level of service security. Also, it offers detailed service access monitoring to allow debugging and SLA conformance checking. Reliable messaging is supported, preventing double execution of retried service requests when connections are suddenly dropped. A service scheduling mechanism based on event triggers can be used to schedule services as a result of other service invocations or as a result of timing events. The scheduling mechanism is leveraged to implement Service workflows that we will discuss in more detail later. In order to create a controllable, fast and reliable run-time we employ a *Staggered Event Driven Architecture* (SEDA)[10]. All potentially blocking operations are performed asynchronously from a FIFO queue. The non-blocking I/O capacity of Tomcat Comet is used to populate a queue of service requests. Service requests can be treated differently, due to e.g. different SLA agreements, by employing priority queues. Navajo scripts are decomposed into decoupled adapter invocations that can be scheduled explicitly.

3 Tipi: a declarative Service Oriented UI framework

Tipi is the UI platform of choice for Navajo based applications. Tipi enables the creation of rich user interfaces, one of the requirements mentioned in the Introduction. Tipi is a combined declarative / procedural language, which is interpreted by the Tipi Run-time. It is very strongly data- and event- driven. The Tipi Run-time has implementations on different platforms, currently on Java Swing and JS/Ajax, using Echo[2]. The language defines the structure of a user interface using a declarative XML language, and procedural event code using either XML, or (Java based) scripting languages. Right now, it supports TipiXml, Ruby and Javascript. The procedural part can call services, and manipulate both the UI-DOM model and the available data (both represented as Navajo Document objects). These blocks of procedural code are attached to events which will fire under certain circumstances. Some examples of these events are `onInstantiate` (when a component is created) `onActionPerformed` (when a user clicks a button) and `onValueChanged` (when a data element gets changed). Tipi uses high level components to link to the data services, and here the data-driven aspect becomes clear. These components have quite a bit of freedom to choose their shape based on the metadata supplied by these services.

[2] http://echo.nextapp.com/site/

Listing 1.3. A data-driven component which will respond to certain service calls

```
<c.panel service="GetTeamSchedule">
    <c.label text="'Some label'"/>
    <c.property propertyPath="'teammatchschedule@0/matchname'">
        <onPropertyChanged>
            <showInfo text="'The value was changed from '+{event:/old↩
                }+'to: '+{event:/new}"/>
        </onPropertyChanged>
    </c.property>
</c.panel>
```

In Listing 1.3, a panel will associate all its data driven children to certain data fields in the `GetTeamSchedule` service. In this case `c.property` is the default data driven component. This component will take on a shape most appropriate for this data. It gets loaded with a Boolean value, it will show itself as a checkbox, if the data is a read-only text field in the next response, it will be just that. Finally, if it is missing completely, it will be invisible. The service provides the metadata, the UI does not. Attached to the `c.property` component is an event definition, which will fire whenever the associated data field changes. In this case, it will show an info message (usually a popup message).

The panel is *listening* to a service called `GetTeamSchedule`. The actual association with the data will take place whenever the service is activated. Any incoming responses, matching the specified Web service, will be delivered to this component. It is completely decoupled from when or why this service was called. It might be the case that there is a explicit call somewhere else in the UI (see Listing 1.4) or when the application uses a pushable connection (for example Jabber), the server can push a `GetTeamSchedule` response to the client, and it will be treated just like other responses.

Listing 1.4. A button to call a service

```
<c.button text='Acquire Data'>
    <onActionPerformed>
        <callService input="{navajo:/SomeInput}" service="'↩
            GetTeamSchedule'"/>
     </onActionPerformed>
</c.button>
```

In Service Oriented Ecosystems, different stakeholders will be re-using the same services. So in general the UI will not exclusively own all the services it is using. That implies that the UI will need to deal with changes. Extra fields may appear or disappear, constraints- and types may change. Usually, UI designers know exactly what data they can expect, but in these circumstances, that is not always feasible. If the services change, the user interface should still function if at all possible. It is important to realize that the UI is part of the eco-system. It should try to give the user the best possible experience, while all its resources (both the platform and the services) are subject to change.

The Tipi platform takes responsibility of all the platform specific details like threading, resource caching and client side storage. The lifecycle management of Tipi applications is managed by the Tipi Application Store, which contains the source files, resources and deployment descriptors. The application store will build a WAR file and install it on a servlet container (for web based applications) or generate a JNLP file (For Swing / SWT based applications). Either way, the Tipi Application Store provides a link to a live application.

4 Navajo Service workflows

To support Task Driven Applications, we introduce a Service workflow mechanism for Navajo. The Navajo framework can be used as an Run-time for Web services. The Run-time supports several features like *hot service deployment*, *authorization*, *logging* and *workflows*. Within the Navajo framework, a workflow is defined as a, service invocation, orthogonal feature that enables a very loosely coupled mode of Service Oriented Application Development. As a result, Service Oriented Applications, especially those offered within a SaaS (Software as a Service) initiative benefit a lot in terms of added *customizability*, *flexibility* and *agility*. We require this feature, because a typical SaaS application can be deployed for different customers each having their own business rules.

A Navajo Service workflow is defined using states and state transitions. An event in combination with a conditional expression can trigger a state transition. A trigger is defined by an event and (optionally) a condition, whenever the event occurs and the condition is valid, the trigger *fires* and a new workflow state will be entered. Within a workflow state, new transitions can be defined declaratively and specific state-dependent services can be invoked; that can trigger another transition in the same or in a different workflow. A Navajo workflow is basically represented by a Finite State Machine.

Examples of commonly used triggers are:

1. A certain moment in time, a unique time stamp.
2. Reoccurring moments in time, e.g. "Every thursday at 22:00" or "Every day at 12:00".
3. A Navajo service invocation. Two different events can be distinguished: just before processing the service and just after processing the service.
4. An internal run-time notification, e.g. "Compilation of Navascript XYZ".
5. An external messaging event, e.g. an incoming Jabber message or an incoming email message.
6. Now or immediate, i.e. for specifying triggers that need to 'fire' immediately.

A Navajo Workflow can be defined using a workflow XML definition. The XML definition specifies all the states and for each state the possible transitions and the activated services.

Each state must have 1 or more transitions defined, with an exception for the acceptance or *null-state*. The null-state is not defined explicitly in the workflow definition because it does not contain any transitions or tasks. It is referred to in the nextstate attribute with the special id `null`. The initial or bootstrap state has special id `init`. The initial state can NOT have any tasks defined, i.e. it can not invoke any services. The initial state defines on which events a workflow instance will be created, i.e. it is the blueprint for creating new workflow instances.

An example of a very simple workflow is shown in Listing 1.5.

Listing 1.5. Definition of a simple workflow.

```
 <workflow>
   <state id="init">
      <transition trigger="navajo:GetTeamSchedule" nextstate="↩
          logresponse"/>
   </state>
   <state id="logresponse">
       <task navajo="response" service="logresponse"/>
       <transition trigger="immediate" nextstate="null"/>
   </state>
</workflow>
```

The defined workflow *listens* to the invocation of the service `GetTeamSchedule`, in fact the transition is fired *after* the completion of the service. It would fire *before* the invocation if the trigger was defined as: `beforenavajo:GetTeamSchedule`. Since the transition is fired after the service invocation, the next state can have access both to the request as well as the response of the service invocation `GetTeamSchedule`.

The transition in the init state initializes a new workflow instance and brings it in the `logresponse` state. In this state a task is defined that invokes the service `logresponse`. The response of the service that triggered the current state is used as a request Navajo; in this case the response of the service `GetTeamSchedule`. The transition in this state immediately brings the workflow to the acceptance state which finalizes the workflow instance.

In many cases workflow transitions may depend on request or response constraints, i.e. the values or existence of certain properties. To support this requirement, it is possible to define conditional transitions as follows:

```
<transition trigger="<transition>" nextstate="logresponse" condition="↩
     <context navajo expression>"/>
```

5 Tiplets: Task driven Tipi

In this section we introduce *Tiplets*. Tiplets are miniature Tipi programs that can be used to create Task Driven Applications.

If a Navajo workflow requires user interaction, a user interface can be pushed to a user or a group of users, effectively asking a user for additional data or a decision. This is realized by a Tiplet, a workflow invoked Tipi instance. It can offer all the power of a Tipi application to assist the user in accomplishing its task.

This is quite different compared to a form based implementation, which only allows a user to enter some data and submit it. In this case it could very well mean that the user has to start their bloated, front-end application, navigate all the menus and look up the data needed to perform the requested service, which was exactly what we wanted to avoid.

In cooperation with the user, the developer can create an application that has precisely the right tools to perform a certain task, and not more. The application can provide a very clear scope to the user, because it is exactly known what the user is trying to do. Furthermore, it enables a loosely coupled development method, since the application is designed and implemented in much smaller parts.

Technically, a Service workflow creates a server side instance of a Tiplet, and pushes a link using (for example) email. The recipient can then use the link to start an application, for as long it is relevant. If the user submits the final result (or actually calls the final Web service as defined in the workflow), the Tiplet will be discarded, and the workflow can move on. This elevates the traditional Client - Server model (with the client being a human and the system being the server) to a peer-to-peer model. This notion of a computer asking a human for info or a decision is not at all new, nor rare. It is what every login dialog, or 'Are you sure?' dialog does. Dialogs typically ask a question they want answered right now and refuse to do much else until you do. Not only are frequent dialogs irritating for the user, it limits their use to short running workflows. In order to be effective for long running workflows, dialogs need to be made asynchronous and persistent. Human tasks need to be scheduled, so the user can choose to perform at a certain moment.

The Tipi Application Store is a possible implementation for scheduling these human tasks. It stores those outstanding requests on the file system as mini Tipi application or *tiplets*. It notifies users of new tasks by sending them a link to the tiplet, and allows them to query their assigned tasks using an RSS feed. From the workflow point of view, calling a Tiplet is like calling a Web service: send a request, and wait for the answer. It might take weeks until a user performs his or her task, or it might never happen. In that case it is the responsibility of the workflow to make sure it won't deadlock (for example by defining a timeout trigger and then aborting the task or assigning it to somebody else).

As a programming model this is much simpler than traditional models, that would typically create a task object at the application level, implementing a specific user interface which allows the user to see those, and then wait until the user starts doing something. It can do the same as a workflow, but its state and flow is much more fragmented, and a lot of the code is application specific, instead of generic.

6 Feasibility

We have evaluated our work within the context of a large SaaS solution called Sportlink[3]. The Sportlink system offers many functional modules that can be used in various sports organization related processes. These functional modules have been set up as generic as possible in order to offer a single SaaS solution for many different sports associations. Different sports associations use different business rules that also require different third party application integration. Within a sport association Sportlink serves a variety of end-users that interact and cooperate in several business processes. The system is used by a large number of sports organizations at different organizational levels, volunteers, athletes

[3] http://www.sportlink.com/

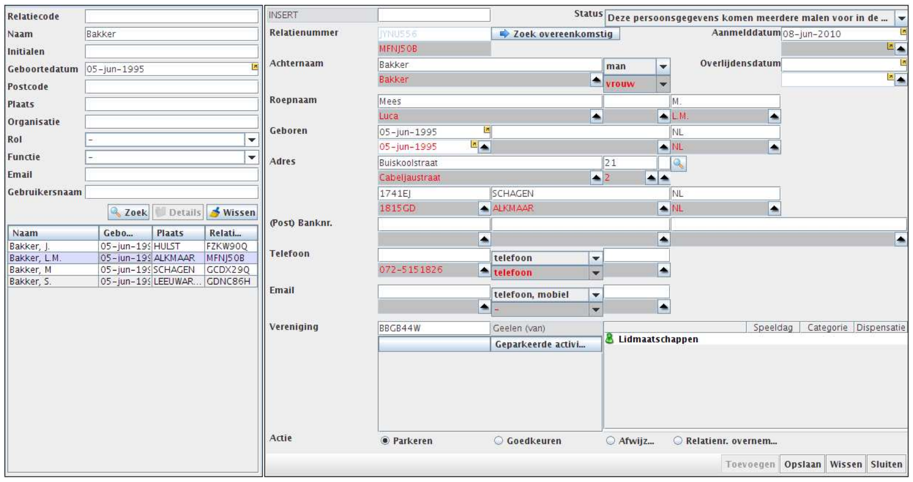
and various other stakeholders with different interests. Each of these stakeholders is involved in parts of the organizational process. The association is in charge of coordinating different tasks, while sub tasks are divided among other entities at lower organizational levels.

The business process that we have implemented using Service workflows, is part of the member administration context: registering new members at the sports association. New members are typically becoming a sports association member via a sports club. The sports club uses Club Management software to enter a new registration. Subsequently, the new registration is synchronized with the main application of the association. Note that all the functions are implemented as Web services. A new registration is checked for business rule conformance by an automatic procedure. The procedure can decide to *park* the registration for manual processing. One of the reasons for parked registration is a potential double member. The workflow shown in Listing 1.6 is responsible for identifying a parked member registration and pushing a Tiplet to the asssocation employee (in `sendparkedtask` state of the workflow).

Listing 1.6. A workflow for notification of parked member registrations

```
<workflow>
    <!-- init: workflow instance is created whenever a new member ↩
         registration has
          resulted in a parked status -->
    <state id="init">
        <transition trigger="navajo:external/↩
            ProcessExternalInsertMember"
                    nextstate="sendparkedtask" condition="[/Result/↩
                        Status] == 'parked'">
            <!-- Create workflow state parameters: not shown -->
        </transition>
    </state>
    <!-- sendparkedtask: Activate task to push Tiplet to an ↩
         association employee
          and immediately go to parked state. -->
    <state id="sendparkedtask">
        <task navajo="request" service="external/ProcessSendTask"/>
        <transition trigger="immediate" nextstate="parked"/>
    </state>
    <!-- parked: Wait until (my!) new member is accepted by employee ↩
         and go to ready state.
          If member registration is not accepted soon enough send ↩
              reminder task. -->
    <state id="parked">
        <transition trigger="beforenavajo:external/↩
            ProcessUpdateParkedMember"
                    nextstate="ready"
                    condition="[/MemberData/UpdateType] == 'INSERT' ↩
                        AND
                               [/MemberData/UpdateStatus] == 'ACCEPTED↩
                                   ' AND
                               [/MemberData/MutationIdentifier] == [/↩
                                   __parms__/WFPersonId]">
            <!-- Create additional workflow state parameters: not ↩
                 shown -->
        </transition>
        <transition trigger="offsettime:5m" nextstate="sendparkedtask"↩
            />
    </state>
    <!-- ready: send an email to a club representative informing him/↩
        her about
         the acceptance of the new member registration -->
    <state id="ready">
        <task service="external/ProcessSendEmail"/>
        <transition trigger="time:now" nextstate="null"/>
    </state>
</workflow>
```

A screenshot of an active Tiplet instance is shown in Figure 1. The Tiplet offers all the required functionality for letting the association user succesfully perform her/his task. For example, being able to search for potentially double members and to query their membership- and personal details. The user can decide to accept or reject the registration. An accepted registration is noticed by the workflow (transition in state `parked`). If the registration is accepted the workflow simply notifies the club user again (in this case via an e-mail) about the accepted registration, and the workflow ends.



Relatiecode	
Naam	Bakker
Initialen	
Geboortedatum	05-jun-1995
Postcode	
Plaats	
Organisatie	
Rol	-
Functie	-
Email	
Gebruikersnaam	

Naam	Gebo...	Plaats	Relati...
Bakker, J.	05-jun-199	HULST	FZKW90Q
Bakker, L.M.	05-jun-199	ALKMAAR	MFNJ50B
Bakker, M	05-jun-199	SCHAGEN	GCDX29Q
Bakker, S.	05-jun-199	LEEUWAR...	GDNC86H

INSERT

Status: Deze persoonsgegevens komen meerdere malen voor in de ...

Relatienummer	JYNU556 / MFNJ50B		
Aanmelddatum	08-jun-2010		
Achternaam	Bakker / Bakker	man / vrouw	
Overlijdensdatum			
Roepnaam	Mees / Luca		M. / L.M.
Geboren	05-jun-1995 / 05-jun-1995		NL / NL
Adres	Buiskoolstraat / Cabeljaustraat	21 / 2	
	1741EJ / 1815GD	SCHAGEN / ALKMAAR	NL / NL
(Post) Banknr.			
Telefoon	/ 072-5151826	telefoon / telefoon	
Email		telefoon, mobiel / -	
Vereniging	BBGB44W	Geelen (van)	

Actie: [✓ Parkeren] Goedkeuren Afwijz... Relatienr. overnem...

Fig. 1. A screen shot of a Tiplet application that is used by the association employee to process a parked member registration

7 Related work

One of the earliest references that mentioned the importance of creating GUIs for Web services is [12]. Their approach was biased towards a web browser and used several artifacts (a GUIDD, WSDL, a Stylesheet and XSLT) to create the final user interface. In [6] an workflow-based method is used to e-mail XForms to workflow participants. In contrast to our approach, they do not have support for pushing full blown, rich, implementation language-agnostics, front-end applications over arbitrary push channels. Also in [11] XForms are used, only this time employing a centralized task server. Push notification using XMPP (Jabber) is used in [20] to initiate a service pull from a client to a service provider.

A declarative UI for service-oriented applications is presented in [8]. Their approach is based on creating several abstract UI models before reaching an actual executable implementation. In comparison, we use Tipi to directly specify executable language-agnostic user interfaces.

A method to programmatically generate XML to support message oriented development is given in e.g. [14]. They augment the standard Java syntax with an additional syntax to support easy formulation of XML processing instructions. An advantage of their approach compared to our Navascript approach is that all the features of Java and XML are directly accessible; in our opinion this can also be a disadvantage for less skilled developers. However, as we have already stated, any service implementation language can be employed within the Navajo framework. Other authors that take, like Navascript, a centralized XML approach for implementing services can be found in [15], [16] and [17]. However, they all support any XML document, making the introduction of XML language semantics (as present in an NXML Document) rather arbitrary, un-constrained and thus difficult for e.g. user interface generation.

The usefulness of Aspect Orientation in the context of Service Orientation is given in [2], [18] and [19]. Our Service Workflows are also aspect oriented, since it executes orthogonally to the normal Web services within the Navajo Run-time. However, we consider our Service workflows to be generalization of Aspect Orientation due to their stateful character. The term *Statefull Aspect Orientation* would be more applicable to stipulate this generalization.

8 Conclusions

In this paper we have explored a Service Oriented Development landscape for setting up Task Driven Front End Applications as means for creating truly Service Oriented eco-systems. In our point of view, the user and the the Web service play a symmetric role in such an eco-system. The two main ingredients for reaching our goal have been: a Service Run-time and language called Navajo and a Declarative UI Run-time and language called Tipi. Both frameworks complement each other in the sense that Navajo can be used to implement headless services, while Tipi can be used to implement the face of these services. Both frameworks are highly focussed to do their respective jobs. On the surface, they are not general purpose languages, but instead high level tools to optimally support the service developer. Both Navajo and Tipi use XML documents that are comprehensible both by humans as well as machines. To support Task Driven applications, two derived ingredients are introduced: Service workflows and Tiplets. Service workflows can be used to introduce state into the sequence of stateless service invocations. By introducing state, business processes, modeled as Service workflows can be defined. These business processes also define the interaction points with human users. In this case, interaction points are again modelled as services, services that are directly pushed to a user. The pushed service can subsequently trigger the instantiation of a mini Tipi application, aka Tiplet, on the device operated by the user.

The main contribution of our work is that we have argumented that it is both sensible and possible to push user tasks in the form of rich user interfaces. Compared to the form based user interfaces used in other work. The feasiblity of our approach has been shown by a real world implementation of a small business process in the context of a Sports Management application.

We firmly believe that the presented Task Driven collaboration approach, enhances the human-machine interaction in such way that the machine is becoming pro-actively helpful to the human user.

References

1. Papazoglou, Mike P., Service -Oriented Computing: Concepts, Characteristics and Directions. In: WISE '03: Proceedings of the Fourth International Conference on Web Information Systems Engineering, IEEE Computer Society, 2003
2. Ranganathan, A. and McFaddin, .S., Using Workflows to Coordinate Web Services in Pervasive Computing Environments. In: ICWS '04: Proceedings of the IEEE International Conference on Web Services, IEEE Computer Society, 2004
3. Ezenwoye, O. and Masoud Sadjadi S., Composing Aggregate Web Services in BPEL. In: ACM-SE 44: Proceedings of the 44th annual Southeast regional conference, pp. 458–463, ACM, 2006
4. Oberleitner J. and Dustdar S., Constructing Web Services out of Generic Component Compositions. In: Web Services - ICWS-Europe 2003, pp. 145–234, Springer Berlin, 2003
5. Waldo, J., Wyant, G., Wollrath, A. and Kendall S., A Note on Distributed Computing, Technical Report, Sun Microsystems, 1994
6. Vélez, Iván P. and Vélez, Bienvenido, Lynx: an open architecture for catalyzing the deployment of interactive digital government workflow-based systems. In: dg.o '06: Proceedings of the 2006 international conference on Digital government research, pp. 309–318, ACM, 2006
7. Hillenbrand, M., Götze, J. and Müller, P., Contract-first service development within the Venice service grid. In: iiWAS '08: Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services, pp. 48–54, ACM, 2008
8. Paterno, F., Santoro C. and Spano, L.D., MARIA: A Universal, Declarative, Multiple Abstraction-Level Language for Service-Oriented Applications in Ubiquitous Environments, ACM Transactions on Computer-Human Interaction, Vol. 16(4), pp. 1–30, ACM, 2009
9. Nestler, T., Design-time support to create user interfaces for service base applications. In: IADIS International Conference WWW/Internet, 2008
10. Matt Welsh: An Architecture for Highly Concurrent, Well-Conditioned Internet Services. Ph.D. Thesis, University of California, Berkeley, August 2002
11. Y.S. Kuo, Lendle Tseng, Hsun-Cheng Hu and N.C. Shih, An XML Interaction Service for Workflow Applications. In: DocEng'06, pp. 53–55, ACM, 2006

12. Kassoff, M., Kato, D. and Mohsin, W., Creating GUIs for Web Services, IEEE Internet Computing (September/October), pp. 66–73, IEEE, 2003
13. Arasjani, A., Ghosh, S., Allam, A., Abdollah, T., Ganapathy, S. and Holley, K., SOMA: A method for developing service-oriented solutions, IBM Systems Journal, Vol.47(3), pp. 377–396, IBM, 2008.
14. Harren, M., Raghavachari, M., Shmueli, O., Burke, M.G., Bordawekar, R. and Pechtchanski, I., XJ: Facilitating XML Processing in Java. In: WWW 2005, pp. 278–287, ACM, 2005
15. Quanzhong Li, Michelle Y. Kim, Edward So, Steve Wood, XVM: A Bridge between XML Data and Its Behavior. In: WWW 2004, pp. 155–163, ACM, 2004
16. One Document to Bind Them: Combining XML, Web Services, and the Semantic Web. In: WWW 2006, pp. 679–686, ACM, 2006
17. Abiteboul, S., Benjelloun, O., Millo, T., The Active XML project: an overview, The VLDB Journal, Vol. 17, pp. 1019–1040, Springer, 2008
18. Braem, M., Joncheere, N., Requirements for Applying Aspect-Oriented Techniques in Web Service Composition Languages. In: SC 2007, pp. 152–159, Springer, 2007
19. J. Niemoller, R. Levenshteyn, E. Freiter, K. Vandikas, R. Quinet, I. Fikouras: Aspect Orientation for Composite Services in the Telecommunication Domain. In: Service Oriented Computing: Proceedings of the 7th International Joint Conference, ICSOC-Service Wave 2009. LCNC, vol. 5900, pp. 19-33, Springer Heidelberg (2009)
20. Weis, T., Saternus, M., Knoll, M., Brandle, A., Combetto, M., Towards a General Purpose User Interface for Service-oriented Context-aware Applications. In: AVI'06, pp. 53–55, 2006