Using Stateful Activities to Facilitate Monitoring and Repair in Workflow Choreographies

J. Eder [+], J. Mangler [#], E. Mussi [*1], B. Pernici [*2]

[+]*Department for Informatics Systems,*
University of Klagenfurt, Austria
johann.eder@uni-klu.ac.at

[#]*Faculty of Computer Science,*
University of Vienna, Austria
juergen.mangler@univie.ac.at

[*]*Dipartimento di Elettronica e Informazione,*
Politecnico di Milano, Italy
[1]enrico.mussi@polimi.it
[2]barbara.pernici@polimi.it

***Abstract*—The repair of faulty processes (workflows, webservice compositions) needs information about the state of the involved webservices. We introduce an architecture where (webservice based) activities and their instances are treated as manageable resources. Based on the WAMO Model our activities provide detailed information about the state they currently hold, possible states they can reach, as well as operations to affect the current state. The strength of this approach is that we can introduce independent repair and monitoring facilities, that utilize a generic way to access information about running activities.**

I. Introduction

Self-healing webservices need to monitor the progress of the execution of a process, detect abnormal situations and diagnose the causes of the failure and repair the process by enable correct continuation of the process.

We define a workflow as a behavioral description, that combines for several webservices how to use the operations described by the WSDL. When such a description is executed it results in a process that holds the runtime information and data associated with a particular set of involved webservices. All information associated with the execution of a particular webservice (e.g. state, return values) is defined as activity. An orchestration is given when the invoked webservices themselves are interfaces to processes.

Typically all information about a particular webservice is collected at design time: a process description represents a static interaction model for a set of webservices. Webservices are seen as collections of operations.

Current webservice implementations are commonly described as stateless, but they often allow for the manipulation of state, e.g. when data values are persistent because of webservice interactions like in WS-BPEL or explicitly through frameworks or standards [1] [2].

Therefore service that provides multiple operations (a consumer can interact with) may quite likely have a internal state (in contrast to the frequent definition of webservices as stateless operations). A standardized way of exposing this state has not yet established in the workflow community.

In our approach multiple webservices are invoked by a controlling process. Whenever an error occurs there is the need to repair the process, which in turn implies the need to fetch information from invoked services at runtime. In our paper we will introduce an architecture that will allow to efficiently provide and mange the information that is necessary to control and repair such processes.

We define webservices as having the following properties:

- webservices have multiple operations [3].
- The operations interrelate on each other in that they return values that are needed as parameters for other operations of the same service.
- Operations can be long running.

Calling an operation results in an activity, which is therefore

- The representation of a running operation.
- An instance of a call to the operation.
- Realized as a new webservice with its own endpoint.

So our approach activities are not only abstract entities associated with the execution of operations of a webservice, but are stateful instances.

The main focus of this work is to control the interaction between a workflow engine and the called webservices. We propose a generic way to query the status of the communication between a workflow and the operations it calls, as well as a generic way to find out which operations are available depending on the status. This mechanism has also to work in the case of failure. So we propose to manage activities as manageable resources.

Activities are described by a set of states, an activity can be in, and operations an activity can perform depending on the states (presented as state graph). These operations include

typical repair operations (e.g. undo and retry. We claim that our approach has the following advantages:

- It provides status information about the interaction between a process and a webservice.
- It provides means to control the interaction (start, stop, abort, undo).
- Information about activities separated and encapsulated from the workflow specific information, and are made accessible through a generic interface.
- The monitoring facility to the system is independent from the business logic.
- The repair facility is independent from the business logic and is able to repair some problems without interfering with the workflow engine.

Our contributions include a way to represent a workflow activity by a fixed set of properties, a way to expose this properties by utilizing the already established WSDM (Web Services Distributed Management) [4] standard, as well as an architecture that implements the above named requirements.

This leads to an environment, in which errors can be detected independently from a workflow process. An independent repair engine can interact with the created activities for possible repair actions, and optionally escalate errors to the workflow engine when repair fails. In our view this leads to increased transparency and simpler, reusable components for repair and monitoring, and to a self-healing system that can transparently handle error cases, that previously to be defined in static workflow descriptions.

We will exemplify our idea with a simple ”airline” example. A flight has to be booked, which includes searching for suitable one, booking it, and in case of a problem, refunding the money (unbooking).

II. Workflow Activities as Stateful Resources

A. Outline

Webservice orchestrations invoke webservices, in that operations are called with a set of parameters[5]. The return value will be used for subsequent calls to other webservices. A webservice choreography, on the other hand, is the interaction protocol between (two) orchestrations (that expose operations through a webservice interface).

A problem with this approach is, that webservices are usually considered as stateless, and error handling is part of a predefined orchestration[5] description. The information and logic necessary to repair the orchestration must be present in the running instance of the orchestration. Current orchestration and choreographies standards [6] [7], focus on describing the interaction or flow of an orchestration or choreography in the sense that they try to provide means to map workflow patterns [8].

We propose to add two issues:

- Whenever an operation is called, it results in an activity, that has a state and a set of operations for interaction.
- The activity is a stateful webservice that lives for the whole lifecycle of an orchestration and encapsulates all information associated with a call to a particular operation it has been spawned for.

The activity is therefore a standardized interface to the functionality triggered by the operation, which of course may internally have its own business logic. The operation acts like a factory [9], that creates activities, which are then the sole communication partners of the orchestration.

With this addition it is possible to access a standardized set of information about the activity, as well as triggering repair actions. Because the activities represent and encapsulate a process, they must live until the end of the orchestration, because they can only be seen as finished, when the orchestration itself is finished, which means that repair of single activities is no longer necessary (because of successful termination).

Failures are the inability of an activity to perform its functionality, this means it cannot deliver the requested service (e.g. flight cannot be booked). Such failures are frequently called semantic failures or expected exceptions [10] as opposed basic failures (e.g. system crash) or application failures (e.g. abnormal program termination). Failures are represented as state changes of the activity (see Fig. 1).

B. The Lifecycle of an Activity

As Ardagna et al. [11] we

> (...) conceive of self-healing behavior as a combination of monitoring and repair capabilities. Our goal is to detect failure during a process execution and apply appropriate recovery actions to let the process successfully terminate.

In [11] a Mediator detects errors and takes steps to “repair” the process. The downside is that the mediator has to have knowledge of the semantic cohesion between the operations, to allow for self-healing. Whereas we try to make operations like undo, redo, compensate, . . . explicit, and thus try to generalize the repair process.

In this section we describe the set of states and operations that are available for the activities that are created as a result to the call of a webservice operation.

In our approach we rely on the foundation laid by WAMO (Workflow Activity Model) [10]. The idea of WAMO is to introduce transactions into the workflow model to handle long running activities. Exceptions that are defined for the activities are automatically handled by the underlying workflow system, when an activity fails. As in WAMO, activities have a common state. Compared to the WAMO model, we deal with the following premises:

- WAMO describes hierarchical structured workflow systems However everything is in-process, where now we deal with a distributed system. Again the activities can consist of sub-activities, but they are not transparent to our system, as they are encapsulated.
- Activities are webservices, that expose an internal state to the orchestration.
- We introduce runtime specific information about states to the dynamic system imposed by the interaction of various partners.

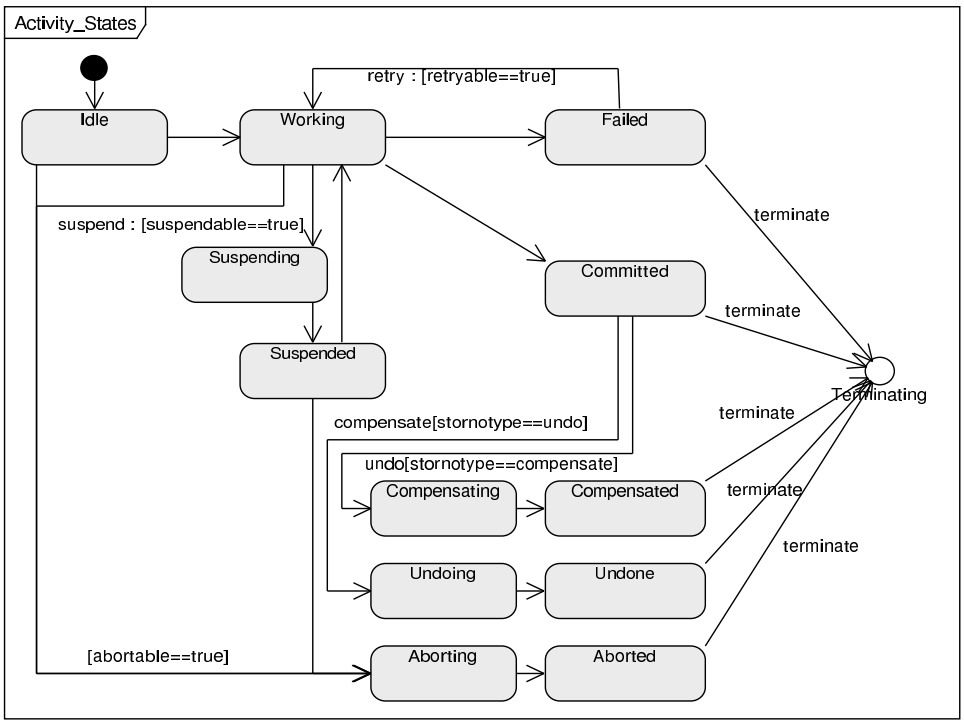

Fig. 1. The States That Can Occur Throughout The Lifecycle of an Activity

Fig. 1 describes a set of states in which an activity can be: *Idle, Working, Failed, Suspending, Suspended, Committed, Compensating Compensated, Undoing, Undone, Aborting, Aborted*

Every activity also exposes a set of properties that are connected to the states, and indicate the possibility of triggering a transition to a different state:

suspendable can be *true/false* in *Working* state. In all other states it is *notdefined*. Suspendable means that the *Activity* can be stopped and resumed (e.g. the shipment can not be suspended).

retryable can be *true/false* in *Failed* state. In all all other states it is *notdefined*. An *Activity* may be retryable when the error that occurred was a temporary one, that is likely to disappear (e.g. connection problems to services that are used internally by the *Activity*).

stornotype can be be either **undo**, **compensate**, **notnecessary** or **notpossible** in *Committed* state. In all all other states it is *notdefined*.

- undo means that there are no consequences from undoing the *Activity* (e.g. all money is transfered back to my account).
- compensate means that some (unspecified) consequences arise from compensating the *Activity*.
- notnecessary is basically the same as undo, but takes zero time (as no method has to be called).
- notpossible means that the *Activity* can not be repaired (e.g. the money is lost).

abortable can be *true/false* in *Idle*, *Working* and *Suspended* state. In all all other states it is *notdefined*. An *Activity* is abortable when the work done so far had no consequences (e.g. no money was transfered yet, or no goods are in shipment so far).

There is a set of operations that can be used to manually trigger transitions to a new state. In the following paragraphs we will describe the lifecyle of an activity, and how the operations can be used.

After being created by a *ResourceManager*, *Activities* are an idle state, waiting to be put either into the working state or into the aborting state. When *Activities* are in the working state it means that they are executing their business logic. In case *Activities* successfully complete they move into the committed state, while if they are not able to complete, or they complete with unsuccessful results, they move into the failed state.

From the failed state, *Activities* can be moved back into the working state by sending a retry message that enables re-executing *Activities* from the beginning. The re-execution is enabled only for those *Activities* that have the property retryable set to true.

During their execution, *Activities* can be paused and moved into the suspended state. Suspended *Activities* can be resumed or aborted. In the former case they are moved back into the

working state, while in the latter they are moved into the aborted state.

Committed *Activities* can be either undone or compensated. Undoing an *Activity* means that it is possible to cancel all the effects it produced without additional costs, while compensating means that it is possible to cancel its effect partially or totally, but with additional costs. *Activities* with the compensate property set to true can be compensated by sending a compensation message that moves the *Activities* into the compensated state. Similarly, activities with the undo property enabled can be undone by sending an undo message that moves the *Activities* into the undone state.

From the point of the *Activity* there is no difference between compensate and undo, as in both operations the underlying functionality is not exposed to the outside. The difference between the two is purely maintained to reflect the terms common at class and process level also at the level of instances.

Since the transitions from one state to the other may be consuming in terms of time and resources used by the *Activities*, we introduced a set of states to represent such transitions. These states are suspending, compensating, undoing, and aborting.

The completion of an *Activity* is achieved by sending the terminate message. That message can be sent when activities are either in the failed, committed, compensated, undone, or aborted state.

III. The Core Architecture

The usage of stateful activities leads to the following Architecture (see Fig. 2). The resource-orientation entails a slightly different approach, when compared to plain webservices:

(1) the orchestration calls an operation
(2) the operation spawns an activity, returns the endpoint of the activity
(3) the orchestration starts the activity
(4) the activity finishes its work and delivers the result to the orchestration
(5) the orchestration can still interact with the activity for possible repair/undo.

This roughly describes the creational pattern [12]. The webservice is seen as factory that spawns activities whenever an operation is called. The advantage of having separate activities is that webservice specific information (the operations and parameters that are needed for the actual problem) can be separated from common information, like the state of the interaction between the process and the activity.

The *ResourceManager* holds a mere stub of an interface when compared to an ordinary webservice. It is a Resource that also includes information about the availability of the service, statistical information about reliability as well as semantic information how the operations of the service depend on each other. This can be useful in repair situations, however this is not in the scope of this paper.

The *ResourceManager* acts as a factory to create multiple *Activities* that represent the whole lifecycle of an activity from

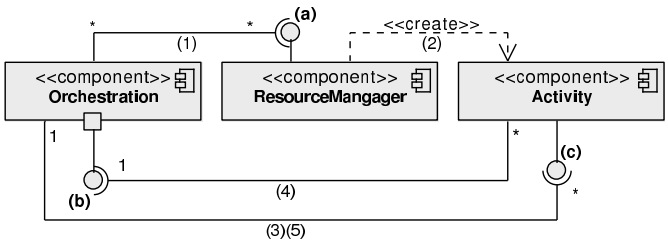

Fig. 2. Architecture Utilizing Stateful Activities

creation to termination. Each *Activity* resource is represented by its own dynamically created endpoint. The *Process* interacts with the *Activity*, solely through either querying its status, receiving status updates via notification messages or triggering certain state changes through a uniform API. A new *Activity* is created for every call to every operation in a certain *ResourceManager*.

As the *Activity* is a resource and potentially long-running, it makes sense that all communication is asynchronous. Polling of the properties is possible, but the normal way of operation should be to subscribe to the *Activity* and receive the state changes as well as the result through notifications.

Of course the integration of such a mechanism is not possible with existing workflow engines capable of execution e.g. standard BPEL [13] code. This integration is no subject of this paper and will be covered in subsequent publications.

IV. Monitoring and Repair Using Webservice Management Standards

In this section we add a monitoring and repair facility to our architecture (see Fig. 3).

(1) The Monitor subscribes to all ResourceManagers. Whenever an Activity is created, its endpoint is advertised to the Monitor. The Monitor subscribes to the Activity through interface (c) in Fig. 2.
(2) The Activity notifies all state changes to the Monitor (as it does to Orchestration).
(3) Whenever *Failure* occurs the *RepairEngine* is invoked, with the endpoint of the Activity and the associated Orchestration. It is also possible for the Monitor to be activated due to timing constraints, ...
(4) The RepairEngine can subscribe to the Activity, query it to get information and trigger operations like retry, undo, compensate, ..., when the repair strategy demands it.
(5) The RepairEngine is notified about all state changes, that possibly occur when after retry, undo, compensate, ... operations.
(6) The RepairEngine is connected to the ResourceManager, in order to be to be able spawn a new Activity to replace a failed one.
(7) The RepairEngine is also connected to the Orchestration to be able to request information about the executed process and to possible repair/modify it.

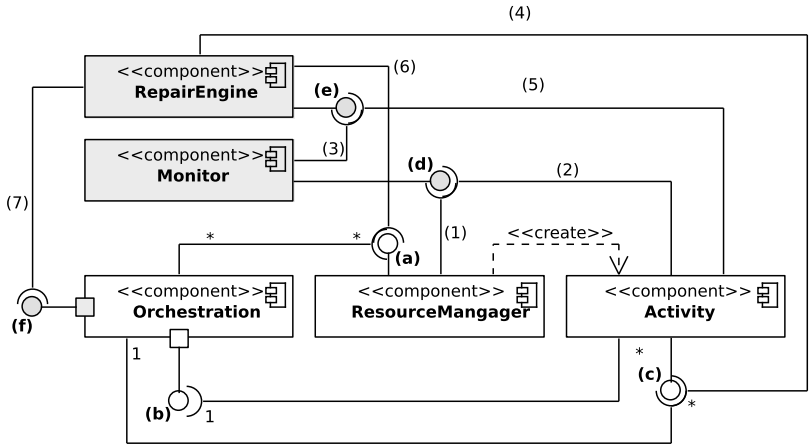

Fig. 3. Monitoring and Repair Facilities

To implement the above sketched architecture, including subscribing to activities, notify services of changes and querying properties about services we rely on the WSDM (Web Services Distributed Management) [4] standard. WSDM (pronounced wisdom) is an OASIS standard to solve the problem of ever growing complexity of business systems by utilizing webservice technology. WSDM is particularly designed for heterogeneous and distributed IT environments, to connect software stacks from different vendors for scenarios where cooperation between different (maybe independent) business entities is necessary. The two main ingredients are the WS Resource Framework (WSRF) [14] and WS Notification (WSN) Oasis Standards [15].

WSRF defines a way to add properties to a webservices, to query which properties are available, and to get and set the value of single properties. It is as well possible to dynamically add properties at runtime.

WSN defines a standard set of operations and their behavior that allows to subscribe to webservice or more specifically to a set of topics, that are also exposed by a webservice.

The WSDM standard makes use of WSRF and WSN and defines a set of properties, that can be queried and subscribed to. Examples for Properties include e.g. Identity, Description, State, OperationalStatus, ... The central idea of the standard is the manageable resource, accessible through a webservice endpoint reference (EPR). The EPR allow retrieve management information, change the state and subscribe to events of a manageable resource.

There are two major initiatives that provide competing technology stacks, that share technical similarities and at least goals with WSDM. The more important is WS-Management which is a DTMF initiative supported by industry heavyweights like Microsoft, Dell, Intel and AMD. It relies on similar but different Standards like WS-Eventing and WS-Transfer. Information about the resources is provided in CIM (Common Information Model) format, which is very generic and allows therefore to represent all sorts of information. There is also JMX (Java Management Extensions) which also features widespread adoptions and is mainly intended for the management and monitoring of applications. The management interface is exposed by MBeans (Managed Beans). Support is included in all main applications servers like JBoss, Tomcat, WebSphere Application Server and Oracle Application Server 10G. The already mentioned WISEMAN has the ability to expose MBeans via WS-Management protocols.

As depicted in Fig. 2 in section III and in Fig. 3 six interfaces are available:

Interface (a) is a WSDM enabled interface, that also contains the operations that spawn the activities.

We make use of the WSDM Advertisement capability. This capability does not define any properties or operations, yet four notification topics. Creation, discovery, destruction and loss of resource connection.

The Advertisement capability is exposed by the ResourceManager because it is having knowledge of the creation or destruction of the activities (resources). External Monitoring and Repair Facilities will subscribe to the Resource Manager to be notified whenever activities are created.

The interface may look like depicted in Fig. 4, class "ResourceManager". *Operations* and *NumberOfRequests* are properties, book and unbook are the operations that are exposed to the orchestration.

Interface (b) is a WSN enabled interface exposed by the orchestration, to receive status updates and the result of the operation (sent by the activity). WSRF specific operations are needed.

Interface (c) exposes is a fully WSDM enabled interface with custom State properties, containing the set of states described in section II-B. The operations and properties are depicted in Fig. 4, class "Activity", and are identical for every activity created for any operation.

Interface (d) is a WSN enabled interface that receives notifications about created activities and their state. The Monitor can invoke the RepairEngine when it receives messages about failure states.

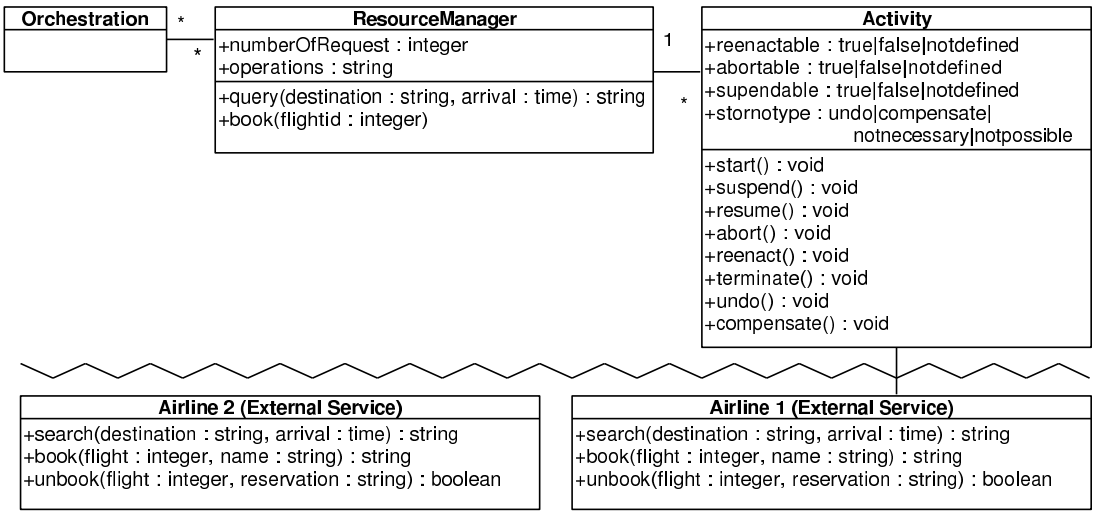

Fig. 4. The API for a Simple Flight Example

Interface (e) is a WSN enabled interface that receives notifications from affected activities only during a repair process. The RepairEngine has to subscribe to this activities when receiving a request for repair from the Monitor.

Interface (f) is a WSDM enabled interface that allows interaction with the workflow engine, like stopping, injecting tasks, change endpoints.

V. A Three-Stage Repair Mechanism

Due the above described mechanisms it is possible to introduce a three-stage repair mechanism to the system. In order exemplify this mechanism we want to add a simple "Airline" example. In Fig. 4 we have several fictional airlines, that expose their booking systems through a simple API. Our high level webservice *ResourceManager* exposes a simplified API and spawns activities that then encapsulate one of the airline APIs. The whole interaction between the orchestration and "Airline" is done strictly through the *Activity*.

The *ResourceManager* does not expose the *unbook* operation as we assume that there is no additional information for un-booking necessary, and therefore the *Activity* can call *unbook* internally when compensate or undo is requested.

Errors can of course occur in the Orchestration, in the Resource Manager and in an Activity, though for now we concentrate on the Activity. When an Activity signals an error to the Monitor, which then invokes the RepairEngine, the repair Engine has the following possibilities:

Stage 1: use the properties and operations provided by the *Activity* to possibly retry the Activity.

Stage 2: escalate the error to *ResourceManager* level. This may include compensating/undoing activities spawned by the *ResourceManager* for a certain orchestration. Repairing on *ResourceManager* level means spawning *Activities* for a different "Airline".

Stage 3: if the above stages fail, there is always the possibility to escalate the error to the orchestration engine. The error may then be dealt with by workflow exception handling, or the RepairEngine can revoke the workflow structure (maybe by user input) to deal with the error.

A possible invocation is depicted in Fig. 5. When the *Orchestration* invokes the search operation a new activity is created, the URL is returned to the orchestration advertised to potential interested 3rd parties (Monitors, ...) trough the WSDM Advertisement facility.

When the *Orchestration* has subscribed to the newly created *Activity* it can start the *Activity* (which was in "Idle", after creation). The *Activity* then signalizes "Working" and finally "Committed" along with the result of the operation. Although it is finished it continues to exist, with the property **stornotype** set to **none**, as storno is not necessary, because a search operation has no side effects at all.

Lets imagine the *Orchestration* involves several *ResourceManagers* and queries them all. In this scenario when the first result returns, it could suspend all other searches, while checking the validity of the result, and later on abort them.

The *Orchestration* internally deals with the result of the search operation, and then calls the book operation of the *ResourceManager*, which again spawns a new MEP of which is then used by the *Orchestration*. For some reason (e.g. the flight is no longer available) after some time period the *Activity* notifies "Failed".

The Monitor catches the notification (it listens to all notifications), and activates the *RepairEngine*. Although the **retryable** property is set to true, the *RepairEngine* may (because of time constraints) decide to give up. As this operation had consequences (the money is already charged from the bank account), the **stornotype** is set to **undo** and *RepairEngine* has to trigger the **undo** operation.

The *Undone* state is safe, the logic inside *orchestration*

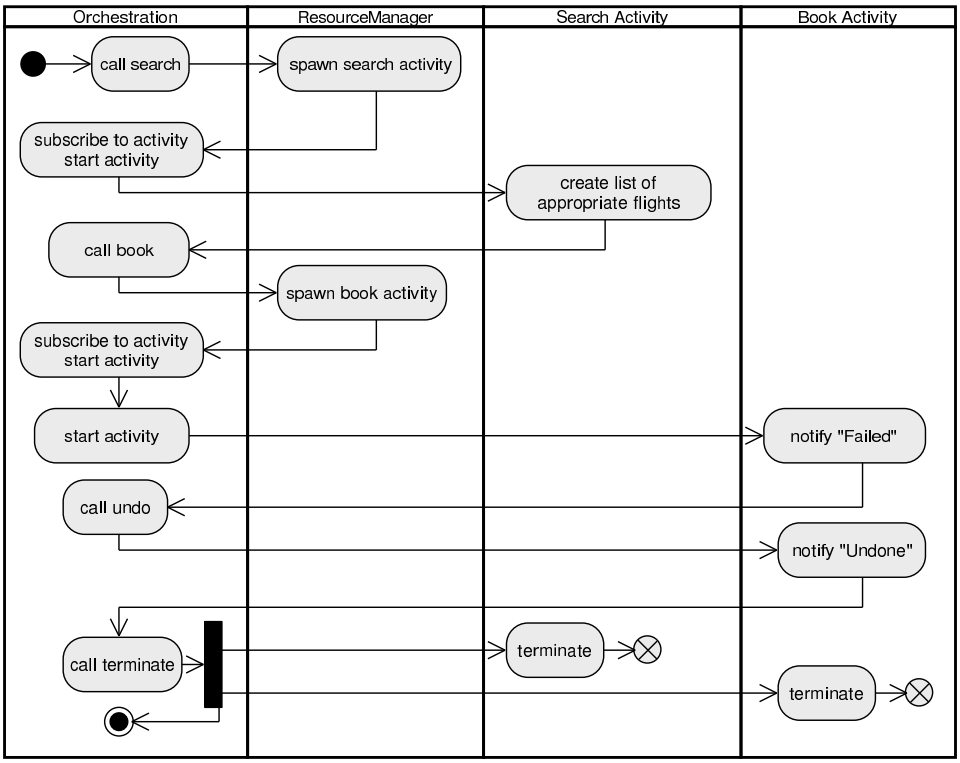

Fig. 5. Booking a Flight

decides to terminate.

We keep *undo* and *compensate* separate mainly for informational reasons. *Undo* is available when after undoing is exactly like before starting. In our case the whole money is returned. *Compensate* on the other hand signalizes that their are bad side effects (like cancellation fees). The *Orchestration* actually has no choice than to call it.

Activities, represented by their own endpoint are not intended to be terminated after their use, but they are destined to stay till the end of the process. So the *Process* has to explicitly call the *terminate* message and clean up all activities after it finishes.

VI. Related Work

We improve upon static workflow models [10] in that we introduce runtime specific information about states to the dynamic system imposed by the interaction of various partners by the means of calls to webservices.

Web Services Choreography Description Language (WS-CDL) [16] and WS-BPEL concentrate on describing possible interactions and possible error situations that can occur during the execution of a workflow. Our approach is not different but complementary to this. WS-BPEL describes a workflow from the point of view of one participant, in our case the *Orchestration*, WS-CDL could very well be used to describe he protocol between *Orchestration* and *ResourceManager*. Our approach focuses on the lifecycle of activities that are created during a workflow, and makes them observable and repairable. Whereas WS-CDL and WS-BPEL focus on information available at design time, we try to make runtime information available.

On the technical side while we use WSDM to make activities observable, there exist also alternatives that could be used. The more important is WS-Management which is a DTMF initiative supported by industry heavyweights like Microsoft, Dell, Intel and AMD. It relies on similar but different Standards like WS-Eventing and WS-Transfer. Information about the resources is provided in CIM (Common Information Model) format, which is very generic and allows therefore to represent all sorts of information. There is also JMX (Java Management Extensions) which also features widespread adoptions and is mainly intended for the management and monitoring of applications. The management interface is exposed by MBeans (Managed Beans). Support is included in all main applications servers like JBoss, Tomcat, WebSphere Application Server and Oracle Application Server 10G. The already mentioned WISEMAN has the ability to expose MBeans via WS-Management protocols.

VII. Conclusions

We presented our approach to facilitate the interaction between Workflow Engines and webservices. This leads to several advantages, specifically when dealing with self-healing systems:

- Monitors and Diagnosing Facilities can easily be added to the system by subscribing to the *ResourceManager* and to the *Activities*.

- Having a uniform representation of activities states is a good starting point for creating a generalized repair algorithm to facilitate self-healing systems.
- The abstraction should sustain the creation of distributed and federated workflow systems, as choreography specific information is no longer intertwined with process specific information.

Future research will elaborate on the issues brought up in this paper. E.g. with our model it becomes possible to easily define Service Level Agreements between the ResourceManager and Services [17], which than can easily monitored and enforced.

ACKNOWLEDGMENTS

This work was partly supported by the Commission of the European Union within the project WS-Diamond in FP6. STREP.

REFERENCES

[1] J. Pasley, "How BPEL and SOA are changing web services development," *IEEE Internet Computing*, vol. 9, no. 3, pp. 60–67, 2005.
[2] F. Leymann, "Web services: Distributed applications without limits," *Business, Technology and Web, Leipzig*, 2003.
[3] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana, "Unraveling the web services web: an introduction to SOAP, WSDL, and UDDI," *IEEE Internet computing*, vol. 6, no. 2, pp. 86–93, 2002.
[4] W. OASIS, "Web Services Distributed Management V1.1," 2006.
[5] C. Peltz, "Web services orchestration and choreography," *Computer*, vol. 36, pp. 46–52, 2003.
[6] W. van der Aalst, "Don't go with the flow: Web services composition standards exposed," *IEEE Intelligent Systems*, vol. 18, no. 1, pp. 72–76, 2003.
[7] A. Barros, M. Dumas, and P. Oaks, *Standards for Web Service Choreography and Orchestration: Status and Perspectives*, pp. 61–74. Springer, 2006.
[8] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros, "Workflow Patterns," *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 5–51, 2003.
[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-wesley Reading, MA, 1995.
[10] J. Eder and W. Liebhart, "The workflow activity model WAMO," in *Proceedings of the 3rd international conference on Cooperative Information Systems (CoopIs), Vienna, Austria*, 1995.
[11] D. Ardagna, M. Comuzzi, E. Mussi, B. Pernici, and P. Plebani, "Paws: A framework for executing adaptive web-service processes," *IEEE Software*, vol. 24, no. 6, pp. 39–46, 2007.
[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Abstraction and Reuse of Object Oriented Design," *Ecoop'93-Object-Oriented Programming: 7th European Conference Kaiserslautern, Germany, July 26-30, 1993: Proceedings*, 1993.
[13] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, *et al.*, "Business Process Execution Language for Web Services, Version 1.1," *Standards proposal by BEA Systems, International Business Machines Corporation, and Microsoft Corporation*, 2003.
[14] W. OASIS, "OASIS Web Services Resource Framework (WSRF) V1.2," 2006.
[15] W. OASIS, "Web Services Notification (WSN) V1.3," 2006.
[16] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, and Y. Lafon, "Web service choreography description language (wscdl) 1.0," *W3C Working Draft*, 2004.
[17] M. Bichier and K.-J. Lin, "Service-oriented computing," *Computer*, vol. 39, pp. 99–101, March 2006.