Service Integration Patterns for Invoking Services from Business Processes

Carsten Hentrich

CSC Deutschland Solutions GmbH
Abraham-Lincoln-Park 1
65189 Wiesbaden, Germany
e-Mail: chentrich@csc.com

Uwe Zdun

Distributed Systems Group
Information Systems Institute
Vienna University of Technology
Argentinierstrasse 8/184-1
A-1040 Vienna, Austria
e-Mail: zdun@acm.org

In a process-driven and service-oriented architecture, services and business processes are typically integrated by invoking services from the activities of the business processes. The software architect and developer must decide how a service is invoked from a business process. In this decision the requirements that result from the business process-driven service orchestration concept must be considered, as well as the functional architecture requirements of the business processes. We present a pattern language that addresses these design issues and represents proven design knowledge for invoking services from business processes.

Introduction

Service-oriented architectures (SOA) are an architectural concept in which all functions, or services, are defined using a description language and have invokable, platform-independent interfaces that are called to perform business processes [Channabasavaiah 2003 et al., Barry 2003]. Each service is the endpoint of a connection, which can be used to access the service, and each interaction is independent of each and every other interaction. Communication among services can involve simple invocations and data passing, or complex activities of two or more services.

In a process-driven SOA the services describe the operations that can be performed in the system. The process flow orchestrates the services via different activities. The operations executed by activities in a process flow thus correspond to service invocations. The process flow is executed by a process engine.

In this paper we address patterns that solve issues related to business requirements when integrating services and business processes. The service perspective and the process perspective generate an environment of conflicting forces that one has to deal with when bringing the two views together. That means, the requirements of business processes must be reflected by the

services, which impacts design decisions of the services. On the other hand, service-orientation and the concept of invoking services by activities in business processes also imply certain requirements on the business process models. In this respect, we will address various types of service invocations and functional architecture design of services to address requirements of business processes and service-orientation in interdependence. The patterns do consciously not address any related aspects such as security and performance issues of service invocations but rather focus in the primary functional issues.

Pattern Language Overview

The patterns and pattern relationships for integrating services into business processes by invoking services from process activities are shown in Figure 1. The SYNCHRONOUS SERVICE ACTIVITY describes how to model a synchronous service invocation in a business process activity. The FIRE AND FORGET ACTIVITY shows how to model a service invocation without any expected output of the service. The ASYNCHRONOUS REPLY SERVICE and the MULTIPLE ASYNCHRONOUS REPLIES SERVICE patterns address how to model service invocations with asynchronous replies in a business process.

Furthermore, the ASYNCHRONOUS SUB-PROCESS SERVICE illustrates how to design a service that only instantiates a sub-process – without waiting in the calling process for the termination of the sub-process. The FIRE EVENT ACTIVITY pattern describes how to model activities that fire certain events to be processed by external systems. The TERMINABLE DELIVERY pattern addresses how to model time-bound dependencies of business processes on required states of business objects.

There are a number of external patterns that play a role in the patterns introduced in this paper. We present thumbnails for these patterns in an appendix at the end of the paper.

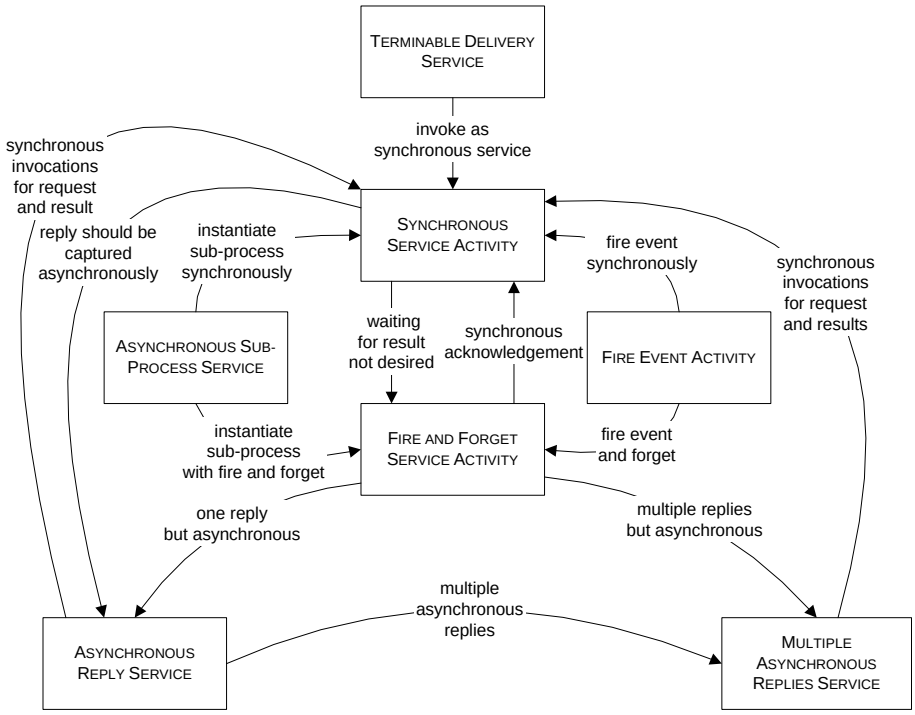

Figure 1: Pattern relationships

Table 1 gives an overview of the problem and solution statements of the patterns.

Pattern	*Problem*	*Solution*
SYNCHRONOUS SERVICE ACTIVITY	Synchronous invocations of a service in a process flow need to be modelled such that the process is able to consider the functional interface of the service and may react on the possible results of the service.	Model a SYNCHRONOUS SERVICE ACTIVITY that depicts the functional input parameters of the associated service in its input data objects and the functional output parameters of the service in its output data objects.
FIRE AND FORGET SERVICE ACTIVITY	Service invocations from a process flow need to be modelled that are not a synchronous blocking call, but rather just placing the service request without waiting for any result to be returned from the service.	Model a FIRE AND FORGET SERVICE ACTIVITY that decouples the request for execution of a service from the actual execution of the service.
ASYNCHRONOUS REPLY SERVICE	Service invocations must be modelled from a process flow that are not synchronous blocking calls, but rather just place the service request and pick up the service result later on in the process flow, analogous to the well-known callback principle.	Split the request for service execution and the request for the corresponding result in two SYNCHRONOUS SERVICE ACTIVITIES and relate the two activities by a CORRELATION IDENTIFIER [Hohpe et al. 2003] that is kept in a control data object.
MULTIPLE ASYNCHRONOUS REPLIES SERVICE	Service invocations need to be modelled from a process flow that are not synchronous blocking calla, but rather just place the service request and pick up multiple replies from the service later on in the process flow.	Extend the ASYNCHRONOUS REPLY SERVICE towards allowing multiple results associated to events representing completed intermediate actions or states of the service.
FIRE EVENT ACTIVITY	Specific states of business processes must be communicated to some unknown target systems and/or functions of systems unknown to the business process need to be initiated by a process activity.	Interpret the states to be communicated and the initiation of external functions as events generated by process activities. Model FIRE EVENT ACTIVITIES that represent event sources, fire appropriate events and depict all distinct states and potential functions to be initiated by their event space.
ASYNCHRONOUS SUB-PROCESS SERVICE	Sub-processes need to be instantiated asynchronously on a process engine without direct support of the process definition language.	Model an activity that invokes an ASYNCHRONOUS SUB-PROCESS SERVICE and which only functionally encapsulates the instantiation of the sub-process on the process engine but not the whole execution of the sub-process.
TERMINABLE DELIVERY SERVICE	The business process expects certain conditions related to business objects to be present by some defined deadline—these conditions reflect the state of the business objects. The process logic thus needs to distinguish whether the business objects related conditions are met in time or not.	Model a TERMINABLE DELIVERY SERVICE that is invoked by a SYNCHRONOUS SERVICE ACTIVITY. The service checks a desired condition with a given deadline of a defined business object and delivers whether the condition is true or false and whether the deadline is reached or not.

Table 1: Problem/solution overview of the patterns

Synchronous Service Activity

Services must be invoked from a process flow.

◆◆◆

Synchronous invocations of a service in a process flow need to be modelled such that the process is able to consider the functional interface of the service and the data dependencies to the service, and may react on the possible results of the service.

In a process-driven SOA, services are orchestrated via a process flow representing a business process. That means services need to be invoked via a process flow. Following the WRAP SERVICES AS ACTIVITY pattern [Hentrich-1 et al. 2006], services are logically related to activities in a business process. A service thus represents the business function related to the corresponding business process activity. However, a service has a functional interface, and in order to invoke a service synchronously from the process flow the business process has to consider the functional in- and output parameters of the service. That means there are data dependencies between the process and the service, or rather more precisely the process activity and the associated service.

This data dependency can be observed on the business process level (also called *macroflow*) and also on the more fine grained IT integration process level (also called *microflow*). No matter what kind of service needs to be invoked (examples for different kinds of services are MACROFLOW INTEGRATION SERVICE and BUSINESS-DRIVEN SERVICE; see [Hentrich-2 et al. 2006]), this data dependency needs to be considered in order to invoke the service. This is a fundamental issue that needs to be addressed in any type of process flow that synchronously invokes a service via a process activity. Consequently, if a service is invoked synchronously and the possible results of the service impact the control flow of the process, the results of the service need to be considered as well.

Hence, both the data dependencies and the service results will influence the design of the process model. As these are rather general issues, it is desirable to resolve them using a general concept for modelling synchronous service invocations in process flows. Figure 2 illustrates the problem.

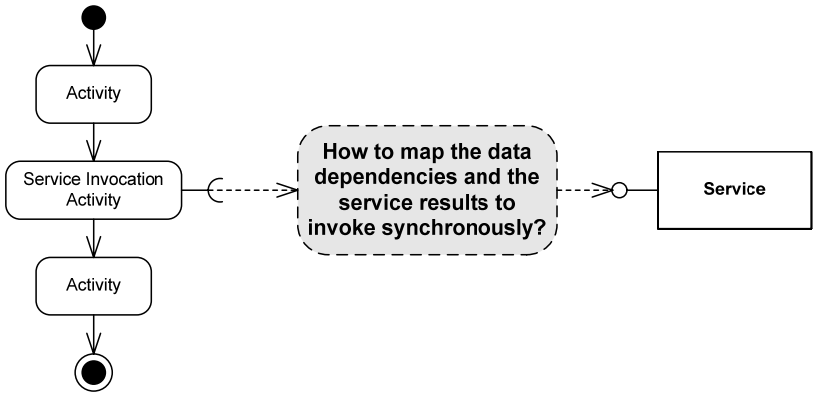

Figure 2: How to invoke services from process flows synchronously?

◆◆◆

Model the service invocation as a SYNCHRONOUS SERVICE ACTIVITY that maps the functional input parameters of the associated service to its input data objects and the functional output parameters of the service to its output data objects. The service is fed with the input parameters from the input objects of the activity, and the output parameters of the service are given back and stored in the output objects of the activity. The process flow, which follows the invoking activity, implements the decision logic to react on the results of the service based on the attributes of the output objects.

Within a process-driven SOA, process flows are executed on dedicated process engines. On the macroflow business process level, those processes are executed on MACROFLOW ENGINES, and on the microflow level, IT integration processes are running on MICROFLOW ENGINES (see [Hentrich-2 et al. 2006] for details). In this context, processes are associated with control data, or rather control data objects to carry the data elements for executing the processes. The decision logic of the processes is based on attributes in these control data objects. The process activities transform these objects by changing their attributes. This is achieved by giving the data objects as input to an activity; the activity changes the contents of the attributes during execution, and sets a new state of the objects as output of the activity. This concept–that processes transform the process control data associated to them– is a general concept of process engines.

The control data object structures need to be designed to depict the requirements of the processes. As a result, these requirements are gathered during the design of the processes themselves, and the control data objects are designed in dependence to the processes in order to capture all necessary requirements. When modelling the synchronous invocation of a service in a process flow, the source for these requirements is the input and output parameters of the service that needs to be invoked.

The control data objects that are used as input for the activity associated to the service must represent the input parameters of the service, as the input data for the service need to be provided by the process. Vice versa, the output parameters of the service need to be represented by attributes of the output control data objects of the activity. In that way, the activity can be functionally mapped to the service interface at the level of data structures. How the actual invocation is performed depends on the techniques provided by the process engine. For example, the process engine might allow the developers to directly invoke a Web Service, or provides some kinds of messaging mechanisms. In any case, the design issue to be solved is the data integration between the process activity and the service, independent of any invocation mechanism. This very design issue is solved by mapping the in- and output parameters of the service to the control data objects used as input and output of the associated process activity.

A prerequisite for executing a SYNCHRONOUS SERVICE ACTIVITY is that the input data for the service must be available in the input data objects of the corresponding process activity. This data is gathered in terms of output data that has been provided by a prior service invocation or by data that has been entered manually in a user interface. As a result, designs at the level of data integration influence the process design, as all data prerequisites for the invocation must be fulfilled by prior process activities.

Consequently, a service invocation in a process needs to be viewed in context with preceding process activities. At the level of data integration often modelling gaps occur that need to be addressed in the preceding process activities (these activities might need to be added, if they are missing). For this reason, the level of data integration must be considered right when designing a process flow containing service invocations.

Furthermore, when modelling a service invocation, it must not only be viewed in context with

preceding activities but also with succeeding activities, as the results of the service invocation will usually be captured by decision logic in the process. Consider for instance the case that the service reports an error—this error needs to be captured by the decision logic in the process and the process needs to react on the error somehow.

The possible cases that influence the path during the process represented by the results of the service invocation must be captured by decision logic of the process right after the invocation. This is achieved by modelling decision nodes based on the attributes of the output data objects that carry the service results. Figure 3 illustrates an example structure of a SYNCHRONOUS SERVICE ACTIVITY.

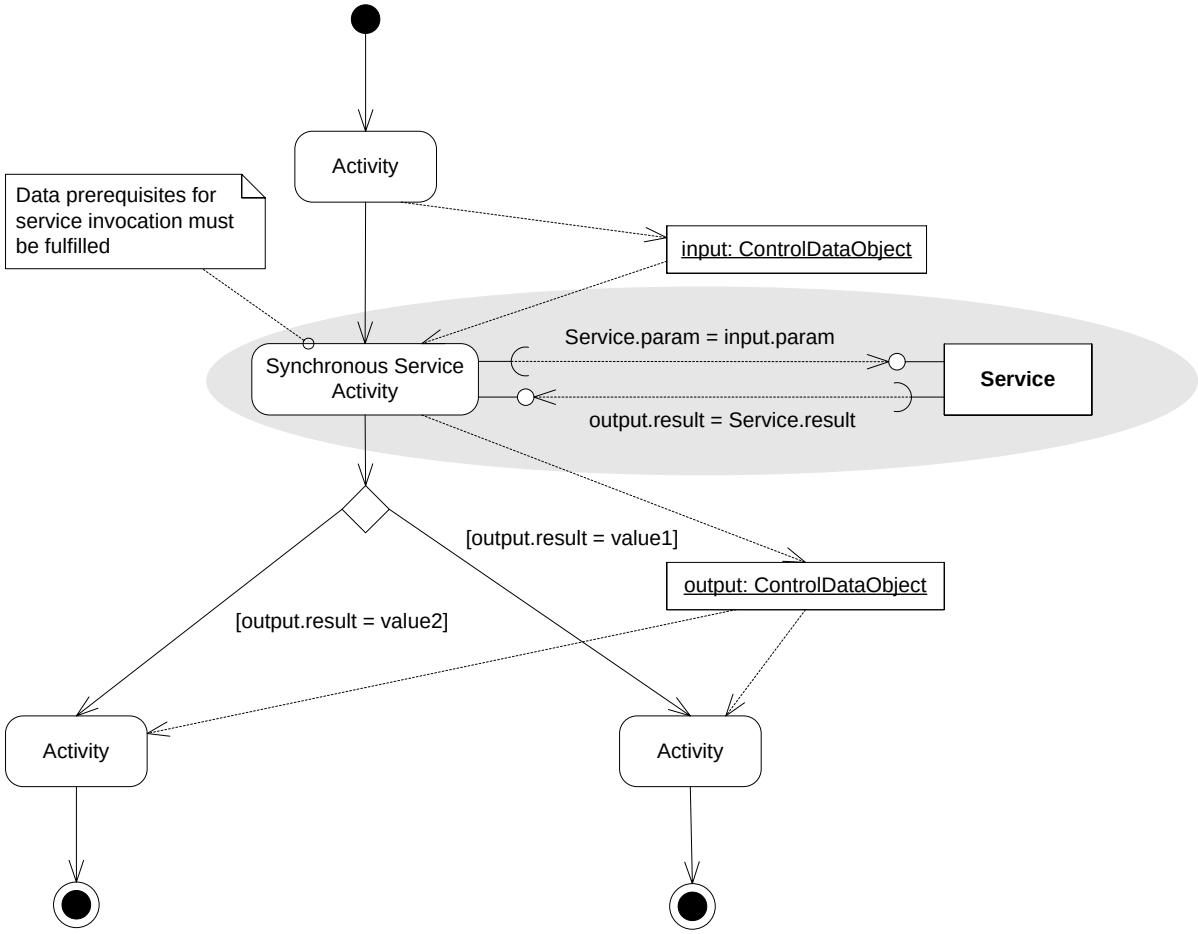

Figure 3: Synchronous service invocation pattern

The control data objects for the processes need to be designed according to the requirements of the invoked services, i.e., the parameters exchanged between a process activity and a service must match. Data integrity should be provided through an integrated design approach for the process and the invoked services. Thus, the process design is aligned with the functionality of invoked services. Vice versa, services can be designed according to requirements of processes.

At the macroflow level, invoked services are very often MACROFLOW INTEGRATION SERVICES, and, at the microflow level, we are usually dealing with invocations of BUSINESS-DRIVEN SERVICES [Hentrich-2 et al. 2006]. For designing the control data objects, the GENERIC PROCESS CONTROL STRUCTURE [Hentrich 2004] pattern can be applied to solve some key versioning issues concerning the data structures. Generic attributes for handling input and output

parameters of services can be defined in the process control data objects that can be reused for various service invocations.

To address error management in the process model, the TIMEOUT HANDLER [Köllmann et al. 2006] and the PROCESS BASED ERROR MANAGEMENT Patterns [Hentrich 2004] patterns are helpful. The TIMEOUT HANDLER addresses how to deal with a timeout situation in case the synchronous service situation does not return any result. The PROCESS BASED ERROR MANAGEMENT pattern shows how to apply generic error management principles in process design.

It is also a matter of service design, what parameters should be given as input and output. Often it is desired to keep the process only concerned with the control aspects of the process but not with the business data. In this case the control data only contains BUSINESS OBJECT REFERENCES [Hentrich 2004]. That means, the actual service being invoked by the process activity only takes a BUSINESS OBJECT REFERENCE as input parameter. The service itself gathers the concrete data from the business object via the reference. Especially MACROFLOW INTEGRATION SERVICES are often designed that way, as this type of service represents a façade that hides the microflow and the corresponding logic for data gathering and transformation for invoking the actual business application services in the backend. If the service reply should be captured asynchronously then the ASYNCHRONOUS REPLY SERVICE pattern applies.

Some known uses of the pattern are:

- IBM WebSphere Process server has synchronous Web services invocation mechanisms based on a architectural model called Service Component Architecture (SCA) which makes use of this pattern. Also implementations based on MQ or JMS are possible, as SCA abstracts from the actual protocol binding of services. BPEL is used as the flow modelling language.
- IBM WebSphere MQ Workflow applies the pattern in its UPES concept to invoke functions from external systems. The example below illustrates an implementation with MQ Workflow. A proprietary flow notation called FDL is used for modelling the flows and the service invocations from a flow model. It is the preceeding product of WebSphere Process Server.
- The BEA Aqualogic component Fuego also applies the pattern in conjunction with synchronous service invocations, based on Web services. BEA uses a proprietary notation and language for modelling the flows and service invocations.
- The FileNet P8 Business Process Manager implements the pattern for invoking Web services from a workflow. At this point in time, FileNet still uses a proprietary modelling language for modelling the flow models. Due to the acquisition of FileNet by IBM it can be expected that FileNet P8 will also move to the BPEL standard in the near future.

These known uses show that the pattern is reflected by many different standard tools and technologies for Business Process Management and workflow. Numerous project implementations exist that are based on these technologies. The SYNCHRONOUS SERVICE ACTIVITY pattern is thus a rather basic pattern related to process-driven SOA.

Example

This example will demonstrate how the SYNCHRONOUS SERVICE ACTIVITY pattern can be implemented. In order to show that the pattern is broadly applicable and not restricted to rather modern SOA technology specifics, such as Web Services, the example will illustrate a more

traditional message-based service invocation with IBM WebSphere MQ Workflow.

MQ Workflow offers a mechanism for invoking services via XML-based message adapters. The whole mechanism is encapsulated in a concept called User Defined Program Execution Server (UPES). The basis of the UPES concept is the MQ Workflow XML messaging interface. The UPES concept is all about communicating with external services via asynchronous XML messages. Consequently, the UPES concept deals with invoking a service that a process activity requires, receiving the result after the service execution has been completed, and further relating the asynchronously incoming result back to the process activity instance that originally requested execution of the service (as there may be hundreds or thousands of instances of the same process activity).

Thus, a UPES is an XML adapter that represents an interface to one or many services. Figure 4 illustrates the process of communication between MQ Workflow and a service via a UPES. Figure 4 gives an overview of the UPES mechanisms.

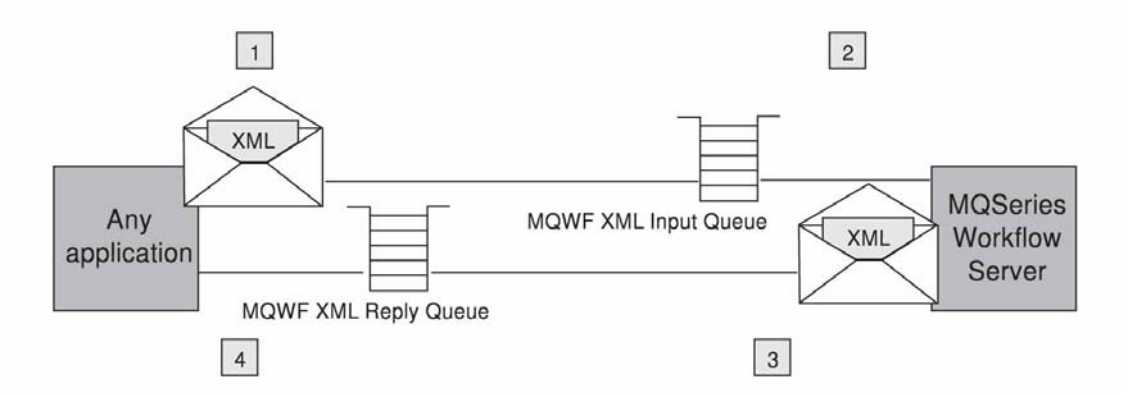

Figure 4: Communication with external services via UPES

1) First, the UPES must be defined and must be related to process activities of process models. Thus, a UPES definition is part of the modelling stage and is included in the final process definition. Basically, from the viewpoint of MQ Workflow a UPES definition consists of nothing more but a message queue definition. An activity related to the UPES thus knows in which queue the XML request must be put.

2) If the execution of a process instance comes to the point where an activity instance is related to a UPES (as defined by the process template), then an XML request will automatically be put in the queue that has been defined for the UPES.

3) The actual UPES implementation is an adapter that listens to the specific UPES queue. It takes the XML message out of the queue, transforms it into a format that can be understood by the service, and initiates execution of the service. The UPES implementation represents a PROCESS INTEGRATION ADAPTER [Hentrich-2 et al. 2006]

4) If the service has finished execution, the UPES implementation will take the result, transform it back into the XML format that can be understood by MQ Workflow and will send the result back to MQ Workflow. MQ Workflow has one specific XML input queue for communicating with UPESs.

5) MQ Workflow takes the return message out of the input queue, relates the result back to the activity instance, and changes the state of the activity instance accordingly.

Modelling the process for this service invocation with WebSphere MQ Workflow is quite straightforward. A control data object is defined to capture the input and output parameters of

the service. This control data object is the assigned as input and as output of the process activity that invokes the service. The process activity is defined as a UPES activity that puts an XML request message in a queue. The result is sent back by the UPES as a result XML message. The request and the return messages are correlated via a CORRELATION IDENTIFIER [Hohpe et al. 2003].

Though the actual communication mechanism via the UPES is asynchronous, it represents a SYNCHRONOUS SERVICE ACTIVITY from the process perspective. The reason why is from the process perspective it is a synchronous communication—the process is waiting for the reply and does not move on to the next step, i.e. it is a blocking service invocation. As a result, the decision whether a communication is synchronous or asynchronous must be seen relative to the perspective of the caller and in context of the architectural layer that is making the invocation. For this reason, the SYNCHRONOUS SERVICE ACTIVITY pattern can be realized even with asynchronous communication mechanisms at the lower abstraction levels.

The following XML fragments show the structure of the request and response messages—the CORRELATION IDENTIFIER named *ActImpCorrelID* is highlighted in the XML. The XML example also highlights that the request message contains the control data object with a customer ID, a currency, and a credit amount that are used as the input parameter for the service. The service being invoked is a credit check, which delivers, based on a customer ID, a currency, and a credit amount, whether a customer is creditworthy for the requested credit amount.

```
<WfMessage>
    <WfMessageHeader>
          <ResponseRequired>Yes</ResponseRequired>
    </WfMessageHeader>
    <ActivityImplInvoke>
          <ActImplCorrelID>FFABCEDF0123456789FF</ActImplCorrelID>
          <Starter>user1</Starter>
          <ProgramID>
                <ProcTemplID>84848484FEFEFEFE</ProcTemplID>
                <ProgramName>FMCINTERNALNOOP</ProgramName>
          </ProgramID>
          <ProgramInputData>
                <_ACTIVITY>Invoke Credit Check Service</_ACTIVITY>
                <_PROCESS>CreditRequest#123</_PROCESS>
                <_PROCESS_MODEL>CreditRequest</_PROCESS_MODEL>
                <ControlDataObject>
                      <CustomerID>4711</CustomerID>
                      <CreditAmount>10000</CreditAmount>
                      <Currency>Euro</Currency>
                </ControlDataObject>
          </ProgramInputData>
    </ActivityImplInvoke>
<WfMessage>
```

The response XML provides the result of the credit check service. The result is contained in the output parameter *Risk* and may have the values "Low", "Medium" or "High". The example XML shows that the credit check result is "Low".

```
<WfMessage>
      <WfMessageHeader>
            <ResponseRequired>No</ResponseRequired>
      </WfMessageHeader>
      <ActivityImplInvokeResponse>
            <ActImplCorrelID>FFABCEDF0123456789FF</ActImplCorrelID>
            <ProgramRC>0</ProgramRC>
            <ProgramOutputData>
                  < ControlDataObject>
                        <Risk>Low</Risk>
```

```
                    </ControlDataObject>
                </ProgramOutputData>
            </ActivityImplInvokeResponse>
</WfMessage>
```

The process model for invoking the credit check service is shown in Figure 5. The process model illustrates that the execution path differentiates whether a low, medium, or high risk has been reported by the credit check service. Note that the notation used in Figure 5 is the WebSphere MQ Workflow visual modelling language. The figure shows a screenshot taken from the modelling tool thus illustrating the actual implementation. The actual language used by MQ Workflow is Flowmark Definition Language (FDL) and the graphical models are translated automatically in this language.

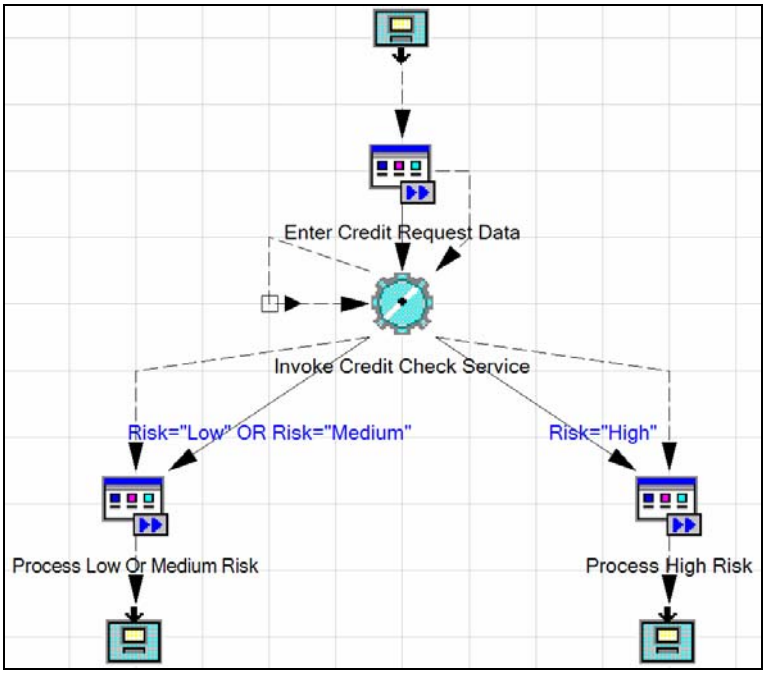

Figure 5: Invoking the credit check service

Figure 6 pictures how the control data object carrying the necessary attributes for invoking the service is assigned as input and as well as output of the activity. The request message is generated automatically by MQ Workflow. The response message of the service contains the risk assessed by the service in the output object. Figure 6 shows the configuration that needs to done in the modelling component of MQ Workflow.

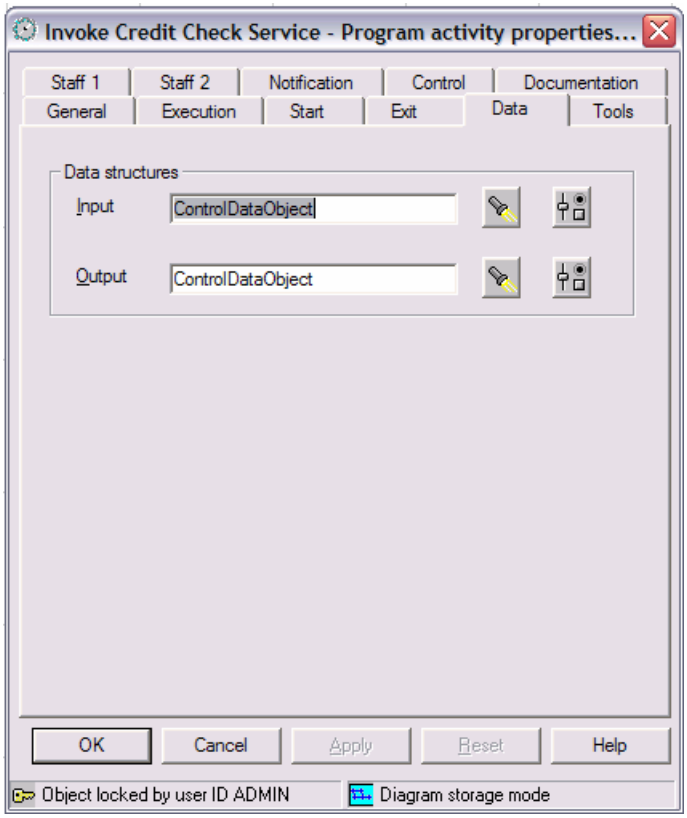

Figure 6: Assigning the input and output control data object

Fire and Forget Service Activity

Services must be invoked from a process flow.

◆◆◆

Service invocations from a process flow need to be modelled that are not a synchronous blocking call, but rather just placing the service request without waiting for any result to be returned from the service.

Depending on the functionality of a service, it is sometimes desired not to wait for the service result. The service request only has to be placed at some point in time, but the process flow needs to continue without considering the result of the service. Often this is the case if the service execution takes a longer period of time, e.g. imagine a batch-oriented function encapsulated in a service and the batch job only runs once a day.

Similar to the SYNCHRONOUS SERVICE ACTIVITY pattern, the data dependencies need to be mapped to provide the right input data for the service by the process activity. For this reason, all the issues identified in the SYNCHRONOUS SERVICE ACTIVITY pattern on providing the right input data for the service by the process activity are the same in this special case. However, how is it possible to invoke the service without waiting for the actual function associated to the service to be executed and not to consider the result of the service at all?

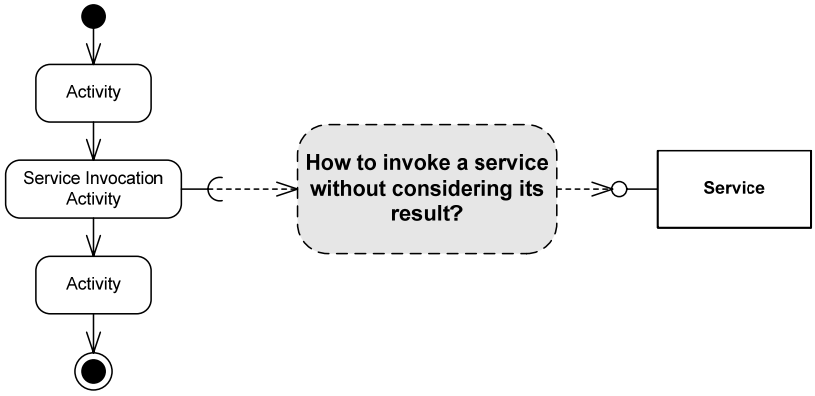

Figure 7: Invoking a service without waiting for the result

◆◆◆

Model the service invocation as a FIRE AND FORGET SERVICE ACTIVITY that decouples the request for execution of a service from the actual execution of the service. Depict the functional input parameters of the associated service request in the input data objects of the invoking process activity. Thus, invoking the service, from the process activity point of view, does actually only mean placing a request for service execution.

The solution separates the request from the execution of the service. This must be done at the process design level and at the remote invocation level. At the remote invocation level, there are two main variants that have both been described as remote invocation strategy patterns:

- An asynchronous execution of the service is performed that fires a request for service execution but forgets about the actual execution. The respective remote invocation strategy that needs to be used is described in the FIRE AND FORGET pattern [Voelter et

al. 2004].

- Alternatively, placing the request may also be understood as a service—a service that does not execute actual business logic but only takes the request for executing business logic. Possibly, this request should be acknowledged. In this case, the solution will be a SYNCHRONOUS SERVICE ACTIVITY in which the service only receives a request for execution and returns an acknowledgment of the receipt of the request. The respective remote invocation strategy that needs to be used is described in the SYNC WITH SERVER pattern [Voelter et al. 2004].

In the SYNC WITH SERVER variant, the result might also contain different options, e.g. request acknowledged or not acknowledged. For instance, if the input values of the request are invalid, the request might not be acknowledged but rather an error message is returned. In the pure FIRE AND FORGET variant, in contrast, the request is only sent, and no acknowledgement or error is returned. Hence, the SYNC WITH SERVER variant can be considered to be more reliable than pure FIRE AND FORGET.

Figure 8 illustrates the structure of the SYNC WITH SERVER variant of FIRE AND FORGET SERVICE ACTIVITY pattern. A service request is placed by invoking a service synchronously. This service accepts a request for actual service execution and returns only an acknowledgement as its output immediately.

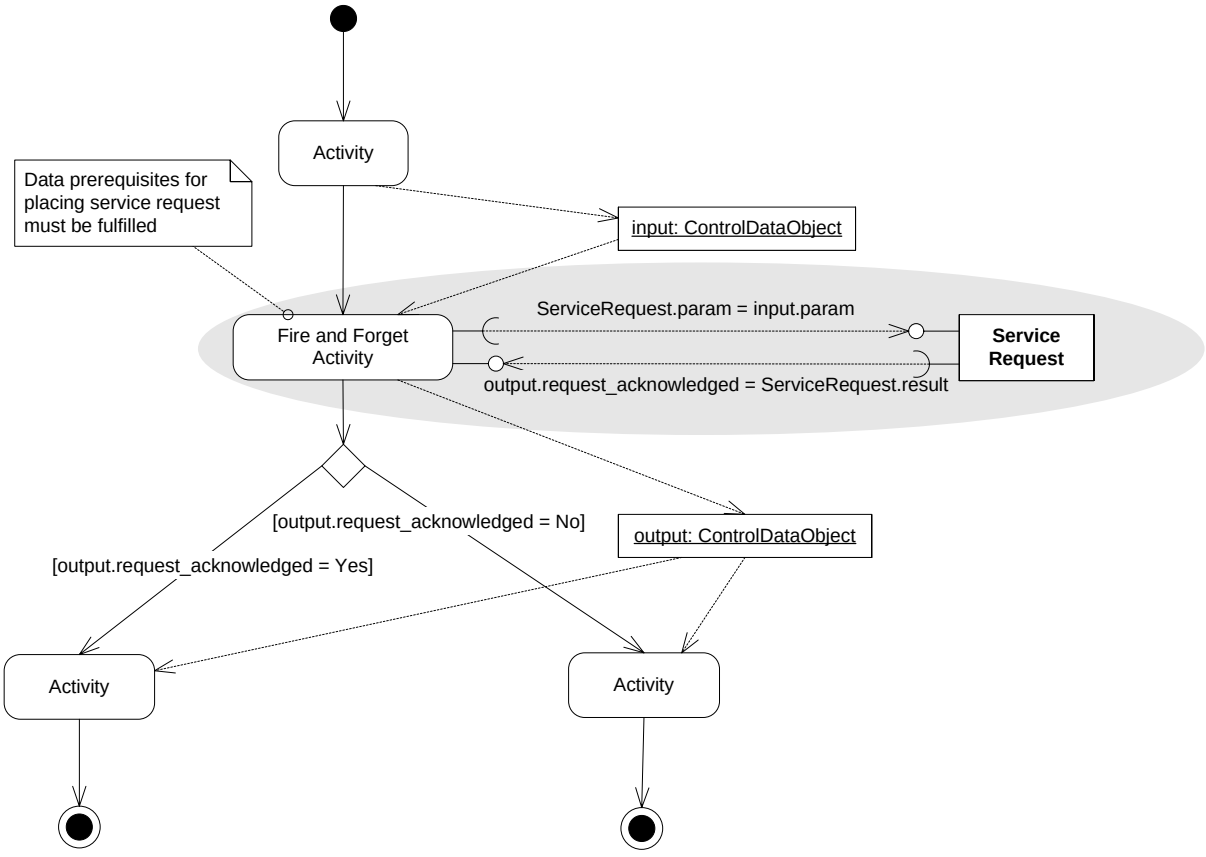

Figure 8: Placing a service request synchronously

The FIRE AND FORGET variant of the FIRE AND FORGET SERVICE ACTIVITY pattern is illustrated in Figure 9. This variant forgets about the acknowledgment of the request and simply places the request neglecting any possible return values. In this case, the process engine used needs to support this mechanism, as the process activity needs to be able to place the request for service execution and the process needs to proceed to the next step automatically. If the engine only supports synchronous invocations, then the service must be designed accordingly, as explained above. In all cases, the right input data must be provided, as already addressed by the SYNCHRONOUS SERVICE ACTIVITY pattern.

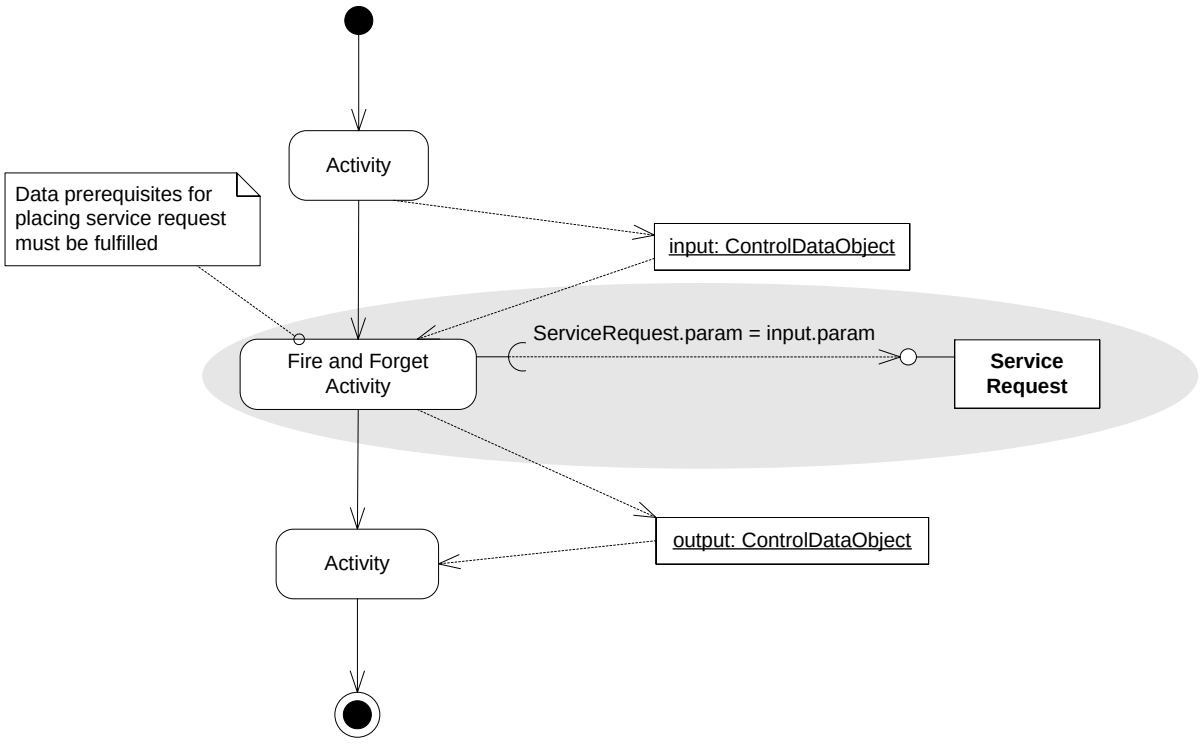

Figure 9: Firing a request and forgetting about the result

The request of a service is decoupled from its execution from the perspective of the invoking process activity. The result of the actual service invocation cannot be determined by the invoking process and possible errors of the execution are not reported. This pattern should only be applied in case the result of the execution is not relevant for further process steps. The process engine used must technically support the two different variants of service invocation mentioned in this pattern. In comparison to SYNCHRONOUS SERVICE ACTIVITY, both variants have the benefit that the process flow can be directly continued without having to wait (block) for a result. In case of the FIRE AND FORGET variant, not even an acknowledgement must be awaited, but for this reason, this variant is less reliable than the SYNC WITH SERVER variant, and should only be used, if best-effort-semantics are tolerable.

The SYNCHRONOUS SERVICE ACTIVITY provides a solution in case an acknowledgement of the service request is necessary. If the result of a service request must be determined asynchronously at a later point in time in the process (callback) then ONE REPLY ASYNCHRONOUS SERVICE ACTIVITY is applied. If there is more than one reply from a service request, then the MULTIPLE REPLIES ASYNCHRONOUS SERVICE ACTIVITY pattern must be used.

Some known uses of the pattern are:

- The known uses given for the SYNCHRONOUS SERVICE ACTIVITY do also apply here, as this is in principle also a very basic pattern related to service invocation from processes.
- GFT's BPM Suite GFT Inspire [GFT 2007] provides a modeller component that uses UML activity diagrams as a notation for modelling the flows. Services can be invoked asynchronously from the flows that can integrate external technologies, such as message brokers.
- In a SOA project for a telecommunications customer, the pattern has been used to define a modelling template for asynchronous service invocations, based on IBM's WebSphere Business Integration Message Broker.
- In a large project on architectural standards in bank in Germany the pattern has been used to fire off service that result in CICS transactions where the process did not need to wait until the actual transaction is finished. Many projects in this bank have been based on this architectural standard.

Example

The SYNC WITH SERVER variant that places the service request synchronously is basically a service design issue of decoupling the service request from its actual function and a variation of the SYNCHRONOUS SERVICE ACTIVITY pattern—we will again take the example of the credit check service presented in the SYNCHRONOUS SERVICE ACTIVITY pattern. The example will show how sending the service request can be implemented with WebSphere MQ Workflow and the UPES communication mechanism.

The only difference, compared to the example in the SYNCHRONOUS SERVICE ACTIVITY pattern is that the return value will not be the actual calculated risk but only an acknowledgement of the service request, as shown in the following XML structure.

```
<WfMessage>
      <WfMessageHeader>
            <ResponseRequired>No</ResponseRequired>
      </WfMessageHeader>
      <ActivityImplInvokeResponse>
            <ActImplCorrelID>FFABCEDF0123456789FF</ActImplCorrelID>
            <ProgramRC>0</ProgramRC>
            <ProgramOutputData>
                  < ControlDataObject>
                        <RequestAcknowledged>Yes</RequestAcknowledged>
                  </ControlDataObject>
            </ProgramOutputData>
      </ActivityImplInvokeResponse>
</WfMessage>
```

Just analogous to the process model in the SYNCHRONOUS SERVICE ACTIVITY pattern, the corresponding process model looks as pictured in Figure 10. The control data object will be assigned as input and output of the process activity invoking the service and the control data object needs to have an attribute defined to report whether the request has been acknowledged or not. The process may than decide on the basis of the value of this attribute which path to go, i.e. whether the request has been acknowledged or not.

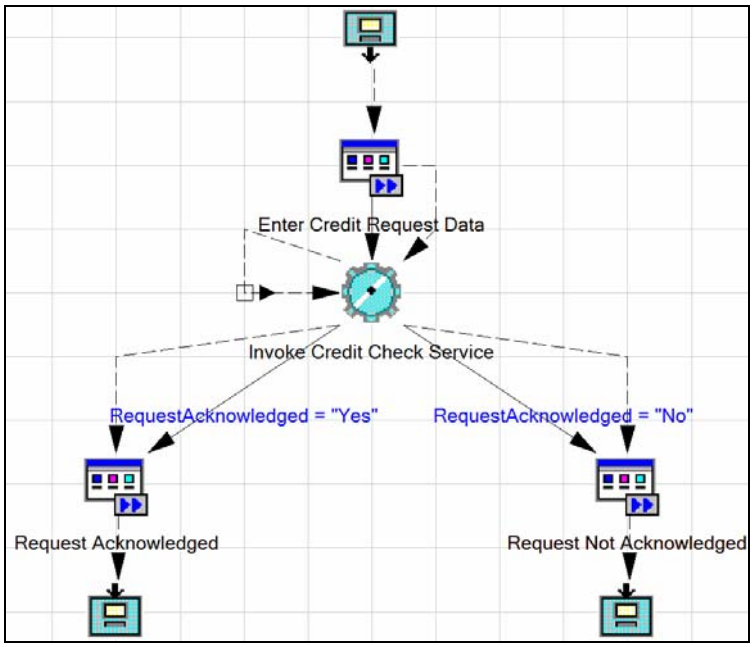

Figure 10: Fire service request synchronously

The FIRE AND FORGET variant of this pattern is about firing a request without considering any return value from the service. In order to apply this variant, the process engine used must also support this unidirectional communication mechanism. MQ Workflow supports the unidirectional communication directly by setting configuration parameters in the process model. First of all the process model looks even simpler as there is no decision logic necessary after sending the service request.

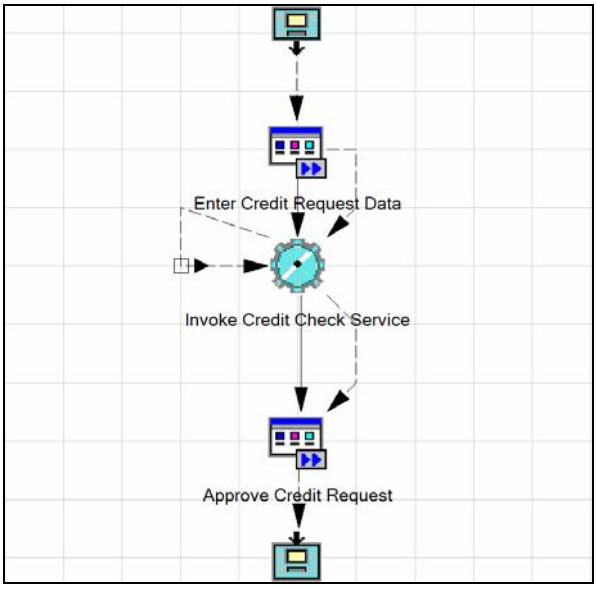

Figure 11: Simply firing the service request

The activity invoking the service must be defined as to expect no reply. This is done in MQ Workflow by setting an asynchronous mode for the UPES communication in the process activity. This is simply done by setting an attribute of the process activity.

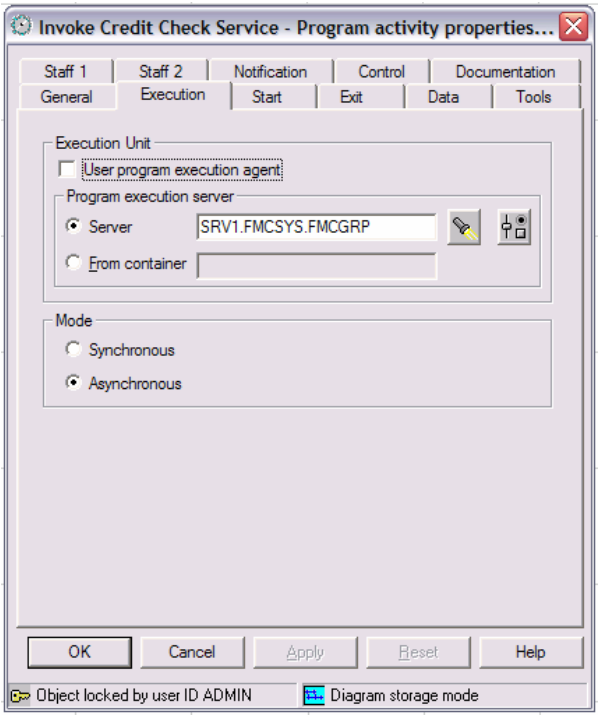

Figure 12: Setting the activity to an asynchronous mode

Asynchronous Reply Service

Services must be invoked from a process flow.

◆◆◆

Service invocations from a process flow need to be modelled that are not synchronous blocking calls, but rather event-based. That is, the service invocation just places the service request and picks up the service result later on in the process flow, analogous to the well-known callback concept.

Sometimes the FIRE AND FORGET SERVICE ACTIVITY pattern is not sufficient as there will be some reply that must be picked up at a later point in time. The process should place a service request, do some other activities in the meantime, and then pick up the result of the previously initiated request at a later point in time. That is, some kind of asynchronous mechanism is required, which only places the service request and the result of the request will be picked up asynchronously at a later point in time. The problem is how to pick up the result later on in the process flow and how to relate a result to a request that has been previously made?

Consider many invocations of the same service have been placed, e.g. by different process instances of the same process that are running in parallel. That means, if a specific process instance wants to pick up a result of one of those invocations, the result must be somehow related to the right request. This relationship is necessary in order not to pick up a result that has been made by another process instance. How can this relationship be realised?

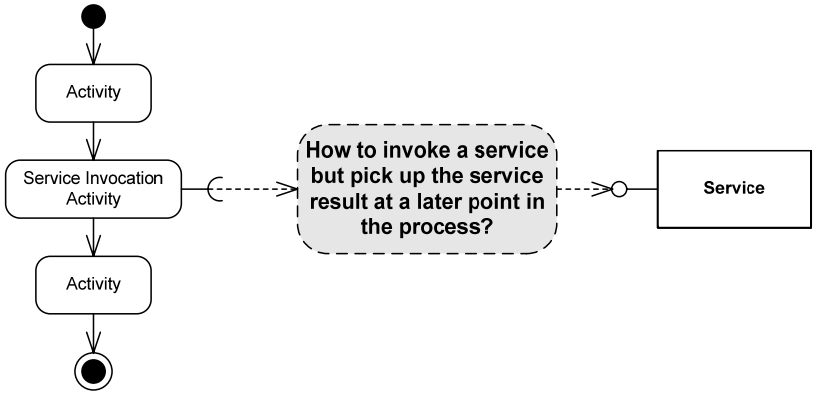

Figure 13: How to realize the callback concept for service invocations in a process flow?

◆◆◆

Split the request for service execution and the request for the corresponding result into two SYNCHRONOUS SERVICE ACTIVITIES and relate the two activities by a CORRELATION IDENTIFIER [Hohpe et al. 2003] that is kept in a control data object. This CORRELATION IDENTIFIER is the output of the first service request, is temporarily saved in a request repository, and is then – later in time – used as an input for the second request that represents picking up the result.

Designing two separate SYNCHRONOUS SERVICE ACTIVITIES enables the separation of the actual service request from picking up the result. The first SYNCHRONOUS SERVICE ACTIVITY represents the actual service request and the second one represents a service invocation that picks up the result. However, to relate a service request and a result, it is necessary to provide a

CORRELATION IDENTIFIER which is generated by the first service. That means, the first service, representing the request, must create a CORRELATION IDENTIFIER that is provided as the output of the service. The identifier is then stored in a control data object of the process and is thus carried along the process activities that may follow.

When invoking the second service to pick up the result, the CORRELATION IDENTIFIER is given as input to the service. The service is thus able to identify the result to the request and to give the right result back to the invoker. For this reason, the *request service* and the *service to pick up the result* have a common functional basis.

The first service registers the requests which are stored in a REPOSITORY [Evans 2004] with their CORRELATION IDENTIFIER that is generated by the service and passed back to the invoker. Then the actual function associated to the service will be executed and the result will also be stored in the REPOSITORY related to the request identified by the CORRELATION IDENTIFIER.

The second service to pick up a result looks into the REPOSITORY to determine the right result based on the CORRELATION IDENTIFIER which has been given as input to the service. If a result is stored in the REPOSITORY for the specific CORRELATION IDENTIFIER then this result is passed back to the invoker. If there is no result stored, then a corresponding error message will be returned. Two possible cases must be distinguished: the CORRELATION IDENTIFIER provided by the service is invalid, i.e. it does not exist in the REPOSITORY, or there is no result yet but the CORRELATION IDENTIFIER is valid. Depending what case applies, a corresponding result message will be returned.

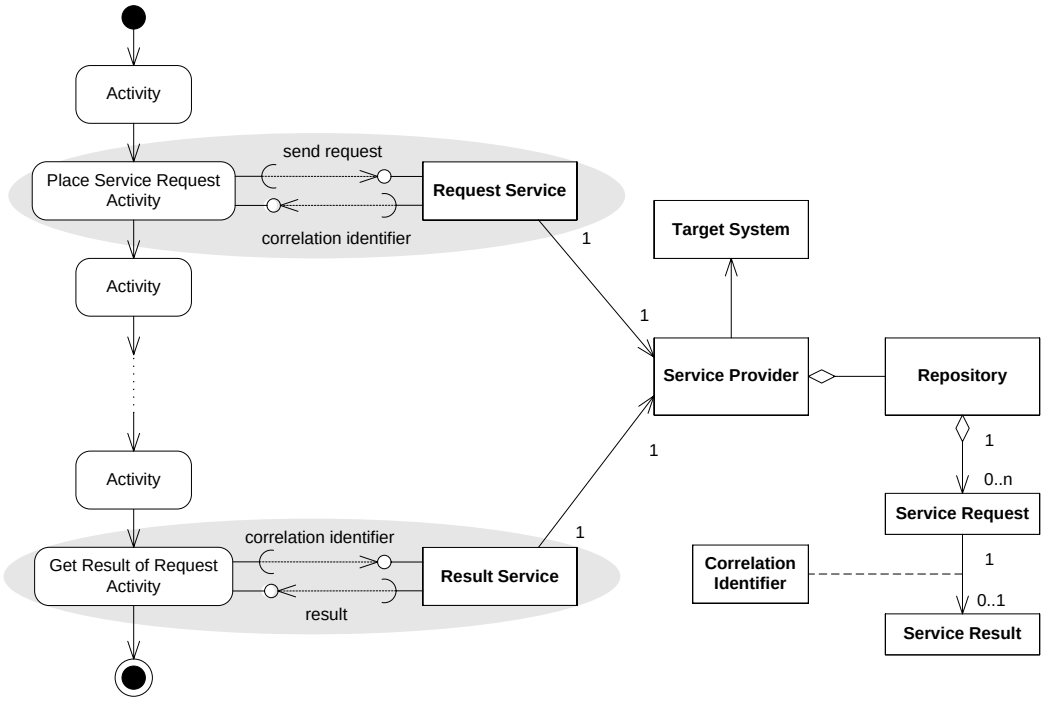

Figure 14: Structure of one reply asynchronous service

Figure 14 shows the structure of the pattern and how the two services are invoked in sequence with other activities between them. Figure 14 also illustrates the functional architecture that is used to maintain the relationship between a request service and a result service via a CORRELATION IDENTIFIER. The following sequence diagrams illustrate in more detail the behaviour that happens when a process instances invokes the two services. The first sequence diagram in Figure 15 shows how the service request is placed. The sequence diagram depicts how the actual function associated to the service request is invoked asynchronously while the activity placing the request already terminates and the process moves on to the next activity. This is indicated by destroying the process activity object placing the request after the service request has been sent.

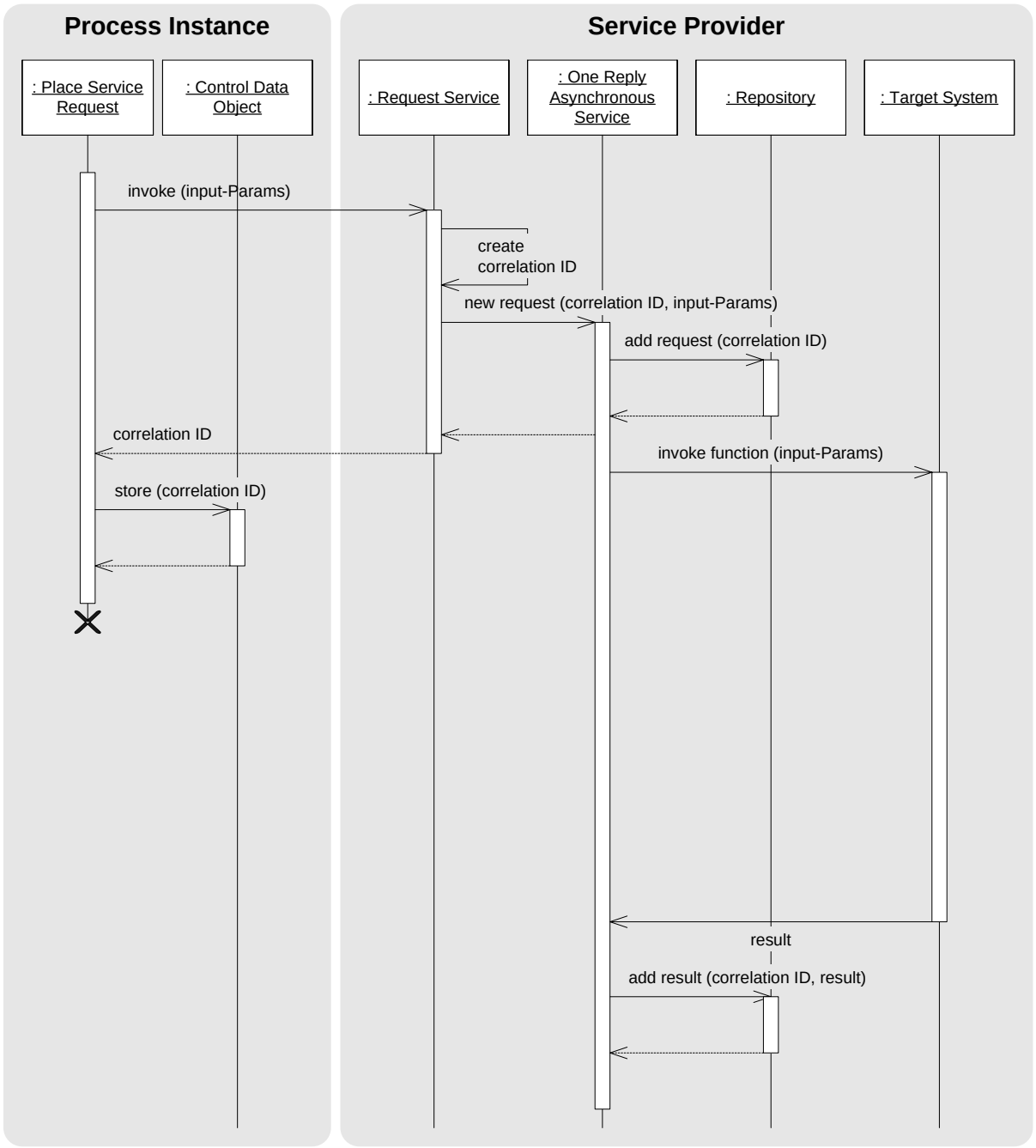

Figure 15: Placing a service request

The second sequence diagram in Figure 16 shows how the result is determined for a request that has been placed before the situation shown in Figure 15 has happend. The diagram illustrates how the result of the service is retrieved from the repository and sent back to the process activity. The right result is determined by providing the correlation ID that has been stored in the control data object.

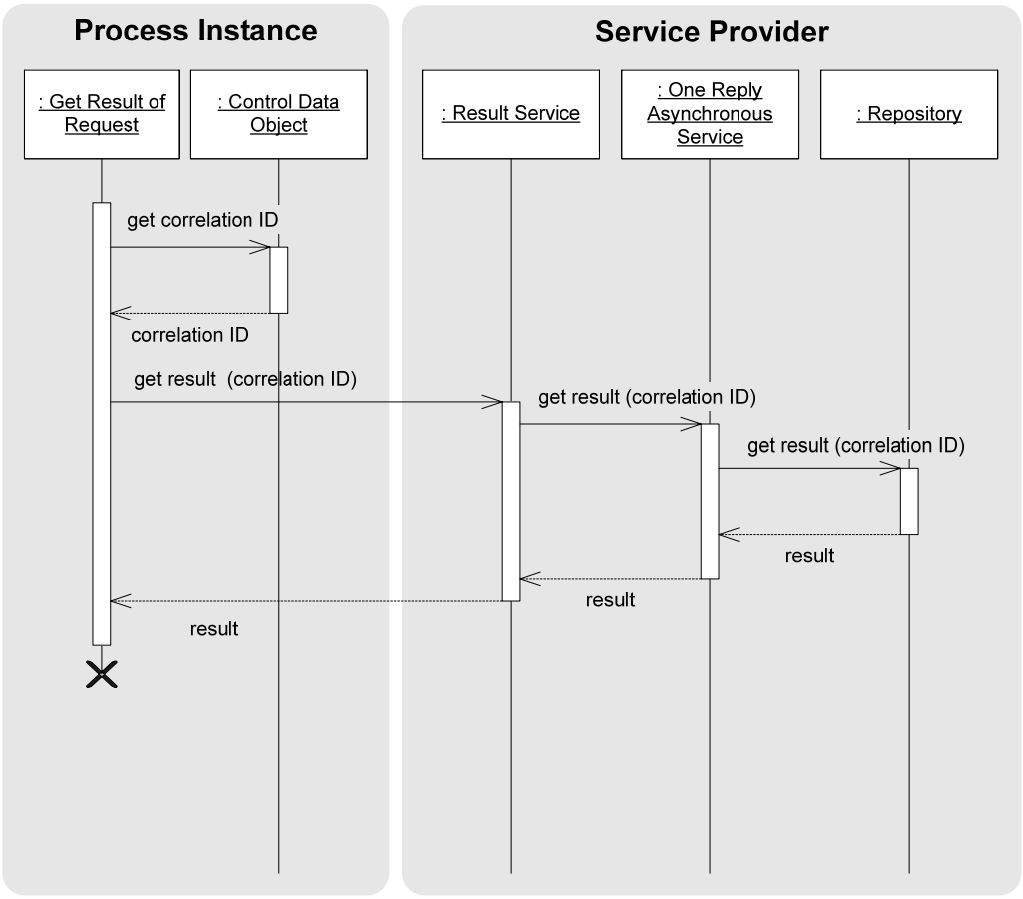

Figure 16: Getting the service reply asynchronously

The ASYNCHRONOUS REPLY SERVICE pattern provides a solution to designing an asynchronous reply on the process design, service design, and functional architecture level. Thus, applying the pattern results not only on the process modelling level but has further architectural consequences that imply additional effort. In particular, a suitable remote invocation strategy must be selected. This is necessary for dealing with situations in which the result service does not deliver a result yet, for instance, because the function associated to the service is not yet completed. There are two main options:

- The process can be actively triggered by a RESULT CALLBACK [Voelter et al. 2004] service invocation that "actively" informs the process about the availability of the result. That is, a real callback is sent as a service invocation to the process.
- Or the process design follows the POLL OBJECT pattern[Voelter et al. 2004]: The process loops the activity invoking the result service (it "polls" it) until the result is available.

In both cases it might be necessary to implement a timeout, e.g. to deal with network failures or other remoting errors.

Once the result is received, the request entry in the request repository should be deleted. The repository possibly needs to implement some cleanup activity, e.g. in case results are not questioned. Some additional logic might be necessary to identify such dead entries in the repository and deleting them, or possible to indicate errors.

As the ASYNCHRONOUS REPLY SERVICE pattern applies the SYNCHRONOUS SERVICE ACTIVITY pattern, the related patterns of SYNCHRONOUS SERVICE ACTIVITY do apply as well to this pattern. Moreover, the request service and the result service mentioned can be realized as MACROFLOW INTEGRATION SERVICES [Hentrich-2 et al. 2006]. The functional architecture part can be realized applying the PROCESS INTEGRATION ADAPTER [Hentrich-2 et al. 2006]. That means, the relationship to the target system can be realized by a PROCESS INTEGRATION ADAPTER that also implements the repository and the management of the relationship between requests and responses.

In many cases the ASYNCHRONOUS REPLY SERVICE pattern must be considered in a larger architectural context concerning several architectural components. For instance, ASYNCHRONOUS REPLY SERVICES are typically used in larger PROCESS-BASED INTEGRATION ARCHITECTURES [HENTRICH-2 ET AL. 2006] where a similar process-logic like the one shown in the sequence diagrams above is implemented as a microflow, and where the repository and the target system are also flexibly accessed as loosely coupled components via BUSINESS-DRIVEN SERVICES [Hentrich-2 et al. 2006]. In this case, the messages related to the service provider part, as depicted in the sequence diagrams, will be service invocations that are orchestrated in a microflow.

Some known uses of the pattern are:

- In principle, the pattern is supported by the process technologies given as known uses in the SYNCHRONOUS SERVICE ACTIVITY pattern as basic support for synchronous services is required. Provided that this support is given the pattern can be implemented.
- In a project in the automobile industry the pattern has been applied to kick-off batch processes from business processes. The batch job runs over night, while in the meantime other process steps have been executed. The result of the batch-job is picked up by the process the following day assuming that the job must be completed overnight.
- In projects related to transaction banking the pattern applies to a similar scenario, where larger or complex transactions are initiated from a business process. The pattern has been defined as a modelling template in an architectural standard of a bank in Germany for those kinds of services related to larger transactions. The technologies used have been an OS/390 based WebSphere MQ Workflow installation in conjunction with a CICS based transaction server.

Example

We will illustrate an implementation with WebSphere MQ Workflow and extend the example of the FIRE AND FORGET SERVICE ACTIVITY pattern towards an asynchronous reply. Imagine the credit check service is provided by an external organisation and it takes about 24 hours to get a result. In this case it actually makes sense to apply the ASYNCHRONOUS REPLY SERVICE pattern and do some other activities in between in order not to waste time. Undoubtedly, this only makes sense if there are sensible business activities possible in the meantime. In this example it is sufficient that the actual result of the service is available at a later point in the process and that it is possible to do further activities without having the result.

First of all, the process model needs to have two activities of service invocations. The first service is placing the request and the second service is picking up the result. The process model

will look as pictured in Figure 17.

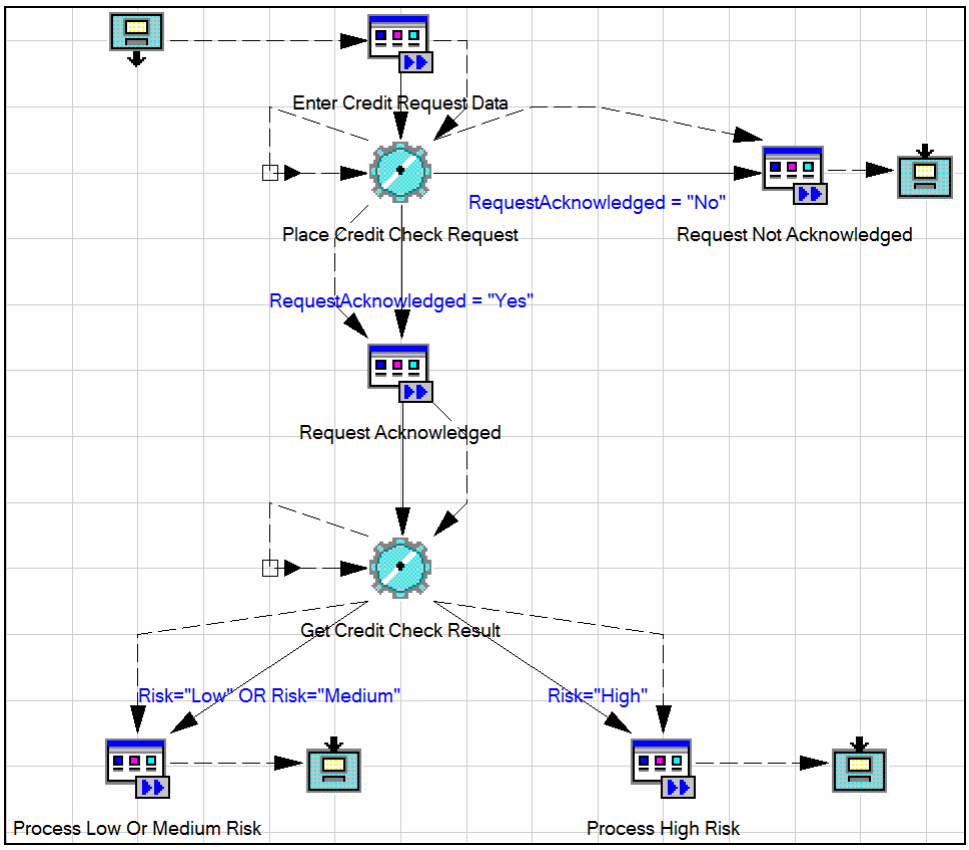

Figure 17: Process model example for one reply asynchronous service

The first service to place the request will have to give back the correlation ID that will be used as an input parameter in the second service invocation. The request XML of the first service is as straightforward as already shown in the SYNCHRONOUS SERVICE ACTIVITY pattern, but the response XML of the first service will be different, as it does not only contain the acknowledgement of the request but also the correlation ID.

```
<WfMessage>
	<WfMessageHeader>
		<ResponseRequired>No</ResponseRequired>
	</WfMessageHeader>
	<ActivityImplInvokeResponse>
		<ActImplCorrelID>FFABCEDF0123456789FF</ActImplCorrelID>
		<ProgramRC>0</ProgramRC>
		<ProgramOutputData>
			< ControlDataObject>
				<RequestAcknowledged>Yes</RequestAcknowledged>
				<CorrelationID>XYAZ4711</CorrelationID>
			</ControlDataObject>
		</ProgramOutputData>
	</ActivityImplInvokeResponse>
</WfMessage>
```

The second service invocation requires the correlation ID as input parameter. For this reason, the request XML of the second service will look as shown below. The response XML of the second service will be the same as shown in the SYNCHRONOUS SERVICE ACTIVITY pattern. It becomes clear how the single service from the SYNCHRONOUS SERVICE ACTIVITY pattern is split up into two services that are asynchronously linked by a correlation ID.

```
<WfMessage>
      <WfMessageHeader>
            <ResponseRequired>Yes</ResponseRequired>
      </WfMessageHeader>
      <ActivityImplInvoke>
            <ActImplCorrelID>FFABCEDF0123456789FF</ActImplCorrelID>
            <Starter>user1</Starter>
            <ProgramID>
                  <ProcTemplID>84848484FEFEFEFE</ProcTemplID>
                  <ProgramName>FMCINTERNALNOOP</ProgramName>
            </ProgramID>
            <ProgramInputData>
                  <_ACTIVITY>Get Credit Check Result</_ACTIVITY>
                  <_PROCESS>CreditRequest#123</_PROCESS>
                  <_PROCESS_MODEL>CreditRequest</_PROCESS_MODEL>
                  <ControlDataObject>
                        <CorrelationID>XYAZ4711</CorrelationID>
                  </ControlDataObject>
            </ProgramInputData>
      </ActivityImplInvoke>
<WfMessage>
```

Multiple Asynchronous Replies Service

Services must be invoked from a process flow.

◆◆◆

Service invocations from a process flow need to be modelled that are not synchronous blocking calls but rather just place the service request. Multiple replies need to be picked up from the service later on in the process flow.

Sometimes even the ASYNCHRONOUS REPLY SERVICE pattern is not sufficient, because there is not only one response from the service but there are multiple possible responses that must be considered. For instance, some services deliver some kind of intermediate results that represent progressing status of the function or task assigned to the service. Often these intermediate results need to be considered from the process point of view. That means, the process may only move on until a certain step after the original service request has been placed and it will only proceed if the previous service invocation has reached a certain intermediate stage that is reported as a status response from the service to the process instance.

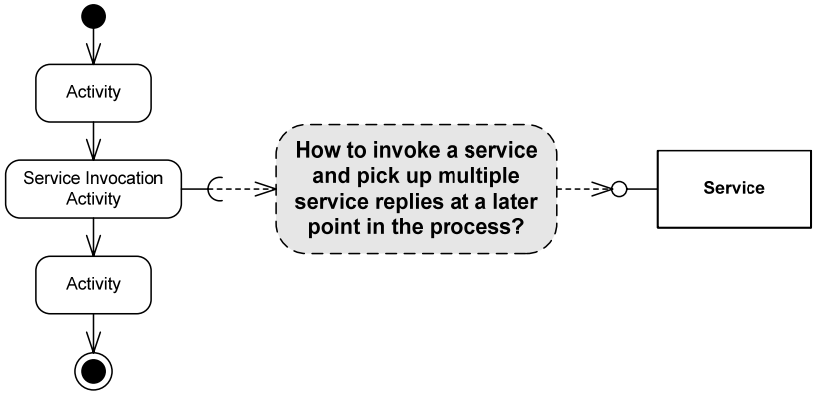

Figure 18: How to realize multiple replies?

The service thus sends several replies which influence the process flow, as each response may have certain results that imply decision nodes in the process. That means, the service may itself be a business process, where intermediate responses report certain states or results of activities in this process. Depending on a state or completed activity of the service, the business process needs to react correspondingly. For example, this can be observed in case the service is a facade of a whole business process running in an external organisation (business process outsourcing) and the service reports several intermediate states of this external business process. The internal business process logic needs to react on the states or completed activities of the external business process.

An example is an ordering process of supplier parts in the automotive industry, where the order is fulfilled by an external supplier. The order may have different states according to the results of completed activities in the outsourced business process, e.g. order accepted or not accepted, procurement finished, shipment date set, shipment initiated, shipment completed, etc. The internal supply chain business process of the automotive company may thus logistically coordinate the procurement of parts ordered from several suppliers, based on the reported states. Figure 19 shows the principle issue of how to synchronize a business process and the process hidden behind the service facade if there are multiple replies to the initial service request.

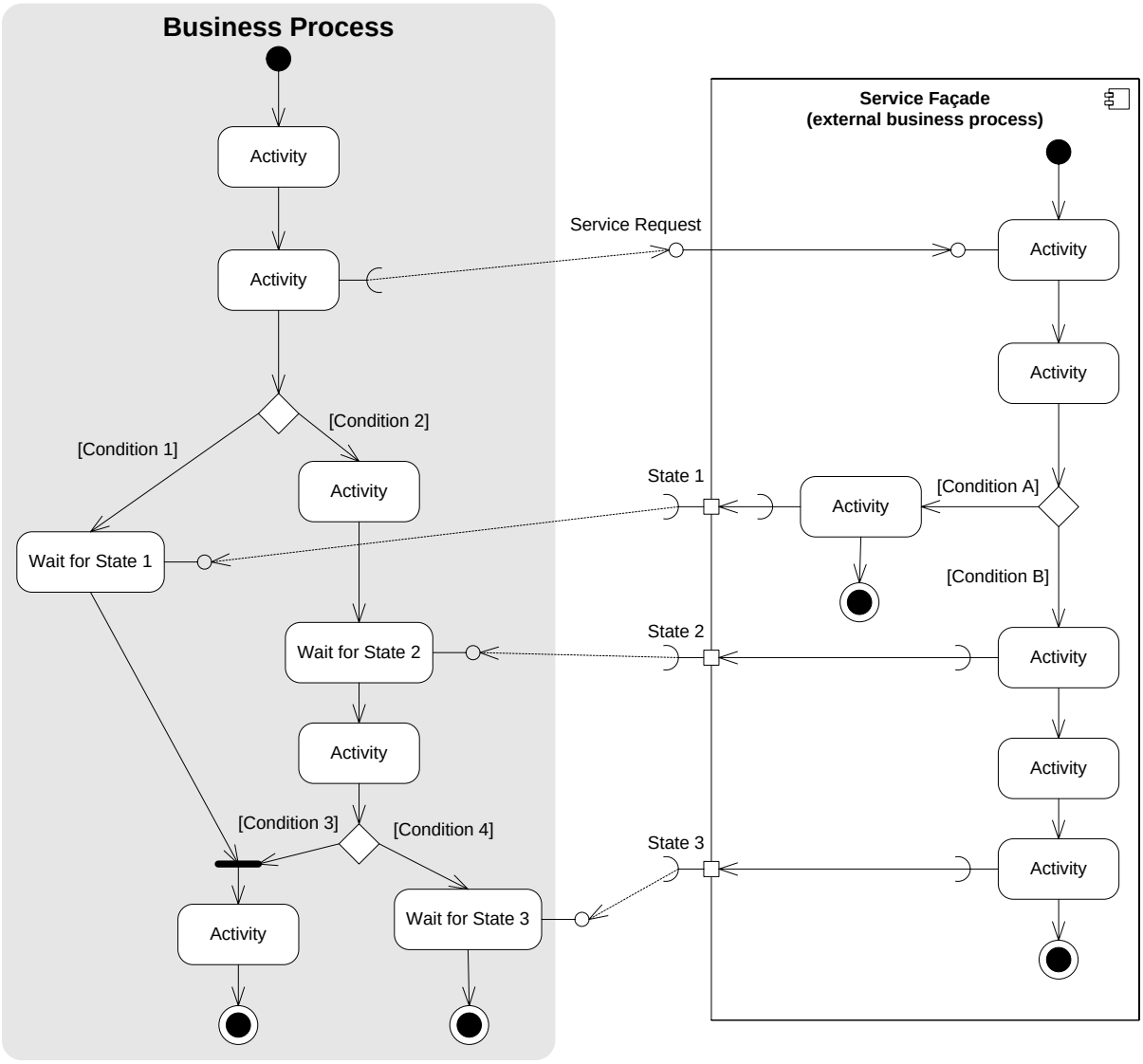

Figure 19: Synchronizing the business process by multiple asynchronous replies

◆◆◆

Extend the ASYNCHRONOUS REPLY SERVICE towards allowing multiple results associated to events representing completed intermediate actions or states of the service. Thus, the result service delivers results based on expected events that are given as input to the result service.

The ASYNCHRONOUS REPLY SERVICE offers nearly all functionality required. The only issue is that it is restricted to only one reply being returned asynchronously by the service. For this reason, the pattern is extended to allow multiple results by introducing the concept of an *event*. The result service allows requesting a result that is associated to a defined event representing a completed intermediate activity or state of the service. The repository stores multiple results to the same correlation ID, where each result is related an event. Events are unique within the space of the correlation ID, i.e. an event may occur only once for a correlation ID. When invoking the result service, the correlation ID and the desired event will be given as input parameters to the service. That way a result can be uniquely identified and is related to an intermediate state of the service. Alternatively, a dedicated result service can be offered for each possible event. The

general structure of the pattern, representing an extension of the ASYNCHRONOUS REPLY SERVICE is pictured in Figure 20.

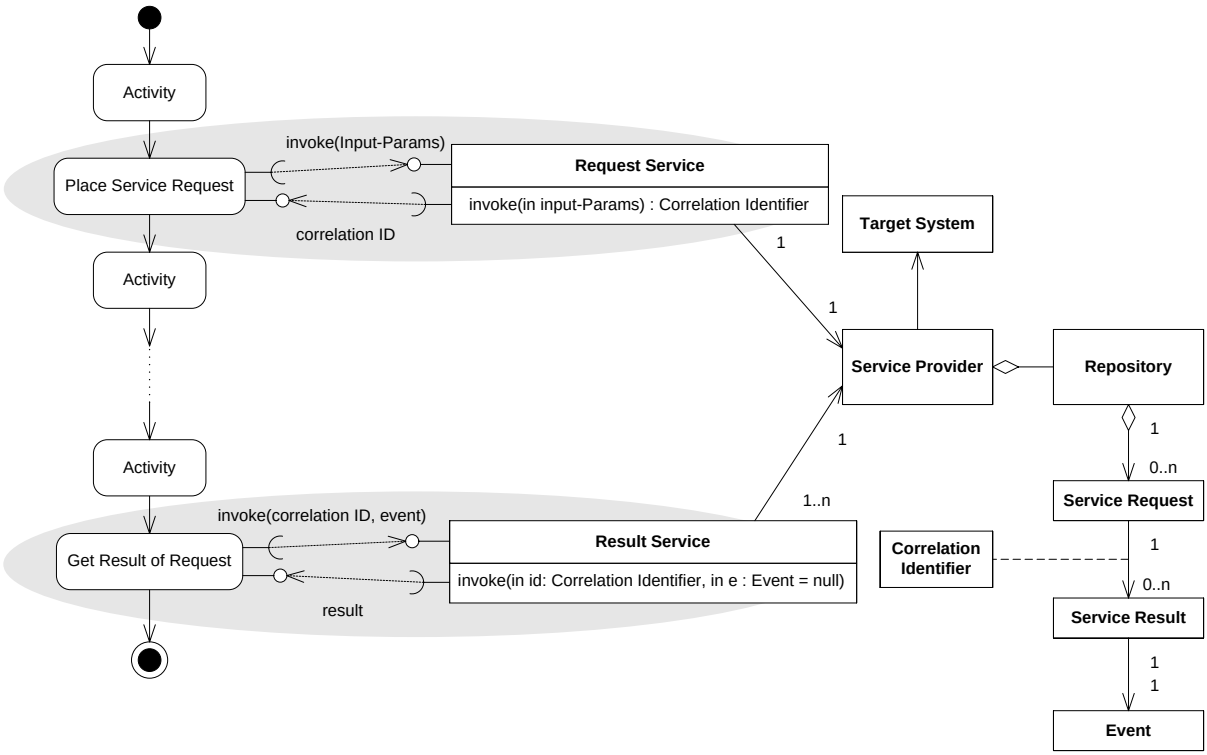

Figure 20: Structure of the multiple asynchronous replies service pattern

The sequence diagram for placing the service request is the same as presented in the ASYNCHRONOUS REPLY SERVICE pattern. If there is a dedicated result service for each possible event, then it is not necessary to provide the event as an input parameter when invoking the result service. In this case the dedicated result service will be implicitly related to a special event. The sequence diagram for invoking a result service is shown in Figure 21. The only difference in this sequence diagram, compared to the corresponding sequence diagram of the ASYNCHRONOUS REPLY SERVICE, is that the event is considered as an input parameter. In order to provide the event information as an input parameter, there must be an attribute that carries the event information in the control data object of the process.

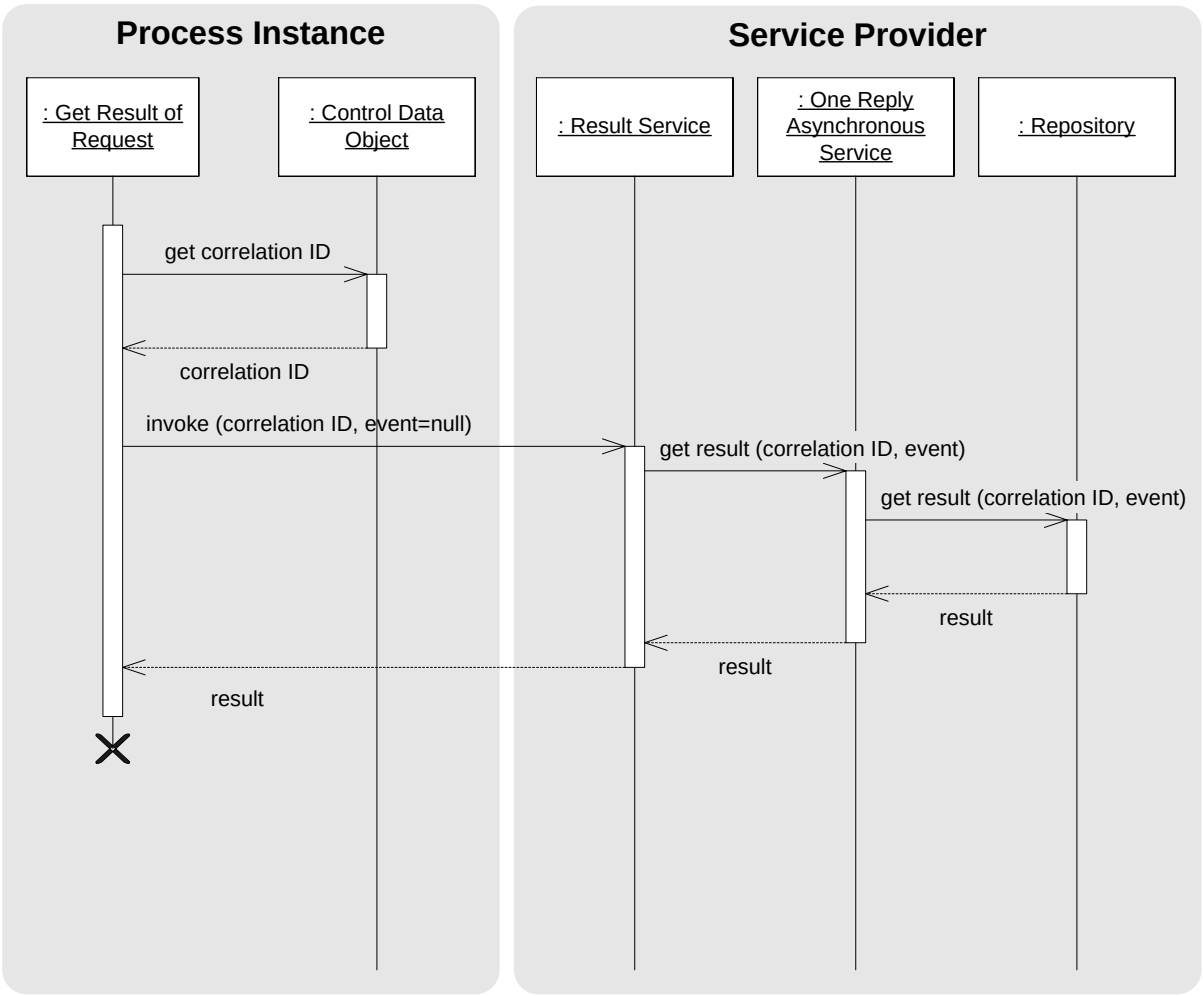

Figure 21: Invoking the result service with an event parameter

The consequences of the MULTIPLE ASYNCHRONOUS REPLIES SERVICE pattern are basically the same as in ASYNCHRONOUS REPLY SERVICE as the MULTIPLE ASYNCHRONOUS REPLIES SERVICE pattern is an extension of it. The only slight difference is that this pattern provides as solution to multiple replies instead of only one reply. Also, the related patterns are the same as in ASYNCHRONOUS REPLY SERVICE.

Some known uses of the pattern are:

- In principle, the pattern is supported by the process technologies given as known uses in the SYNCHRONOUS SERVICE ACTIVITY pattern as basic support for synchronous services is required. Provided that this support is given the pattern can be implemented.
- The pattern has been used in projects in various industries such as telecommunications and automotive to implement loosely coupled coordination of different independent departments and even external organisations. The progress of the processes in the different units has been coordinated by exchanging intermediate states according to the pattern. For instance, in a telecommunications project in Spain the pattern has been used to communicate with an external cable provider to report on progress of an installations process.
- IBM's order management and invoicing solution Webshop is designed to offer a service interface based on Web services that allows querying intermediate states of an order.

These services can be invoked from any workflow tool that integrates Webshop according to the pattern.

Example

We will extend the credit check example from the ASYNCHRONOUS REPLY SERVICE pattern. We assume that the credit check service is still provided by an external organisation and that it takes about 24 hours to complete. However, this time we assume that there is one intermediate completion of an activity reported. This intermediate state provides information whether the requester is on a blacklist or not. Thus, we will not know the final risk factor but we will get information whether the candidate is on a black list or not. Depending on this information the business process may already take some other steps that can be done before the final result is delivered by the service.

The process model will now have to show three service invocations. The first invocation represents the original service request. The second invocation checks whether the requestor is on the blacklist and the third invocation retrieves the final risk factor. The process model implemented with WebSphere MQ Workflow will look as pictured in Figure 22.

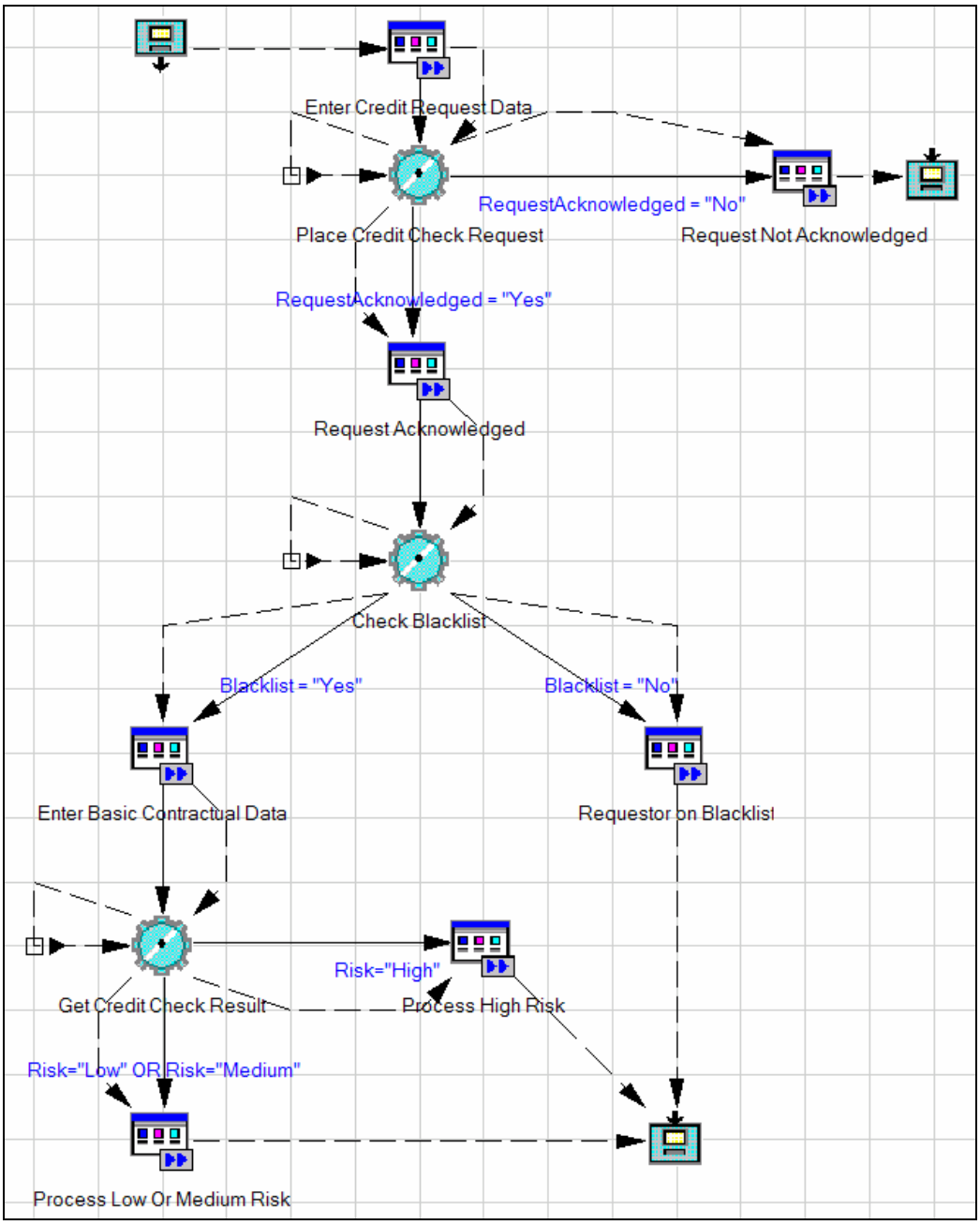

Figure 22: Getting intermediate results asynchronously

From this process model point of view it looks at the first sight as if there are three independent service invocations. However, the invocations are all linked together at a deeper conceptual level by the correlation ID and moreover by the functional architecture of the services. The XML structures of the service placing the request will not be different to the structures illustrated in the example of the ASYNCHRONOUS REPLY SERVICE pattern. The two result services need to consider the event information. For this reason, the input XML of the blacklist check service will look like this:

```
<WfMessage>
      <WfMessageHeader>
            <ResponseRequired>Yes</ResponseRequired>
      </WfMessageHeader>
      <ActivityImplInvoke>
            <ActImplCorrelID>FFABCEDF0123456789FF</ActImplCorrelID>
```

```
            <Starter>user1</Starter>
            <ProgramID>
                  <ProcTemplID>84848484FEFEFEFE</ProcTemplID>
                  <ProgramName>FMCINTERNALNOOP</ProgramName>
            </ProgramID>
            <ProgramInputData>
                  <_ACTIVITY>Check Blacklist</_ACTIVITY>
                  <_PROCESS>CreditRequest#123</_PROCESS>
                  <_PROCESS_MODEL>CreditRequest</_PROCESS_MODEL>
                  <ControlDataObject>
                        <CorrelationID>XYAZ4711</CorrelationID>
                        <Event>Blacklist</Event>
                  </ControlDataObject>
            </ProgramInputData>
      </ActivityImplInvoke>
<WfMessage>
```

The output XML will deliver the result of the blacklist check. In order to report this result to the process instance, the control data object will need an attribute to carry that information. That way the decision logic can be defined according to the result of the blacklist check (see also Figure 22). The result XML from the service sent to MQ Workflow is shown below.

```
<WfMessage>
      <WfMessageHeader>
            <ResponseRequired>No</ResponseRequired>
      </WfMessageHeader>
      <ActivityImplInvokeResponse>
            <ActImplCorrelID>FFABCEDF0123456789FF</ActImplCorrelID>
            <ProgramRC>0</ProgramRC>
            <ProgramOutputData>
                  < ControlDataObject>
                        <Blacklist>Yes</Blacklist>
                  </ControlDataObject>
            </ProgramOutputData>
      </ActivityImplInvokeResponse>
</WfMessage>
```

The second service result invocation finally queries for the final event that reports the risk factor. For this reason, the input XML needs to specify that the questioned event is the calculated risk value. It will also show the same correlation ID like the blacklist service.

```
<WfMessage>
      <WfMessageHeader>
            <ResponseRequired>Yes</ResponseRequired>
      </WfMessageHeader>
      <ActivityImplInvoke>
            <ActImplCorrelID>FFABCEDF0123456789FF</ActImplCorrelID>
            <Starter>user1</Starter>
            <ProgramID>
                  <ProcTemplID>84848484FEFEFEFE</ProcTemplID>
                  <ProgramName>FMCINTERNALNOOP</ProgramName>
            </ProgramID>
            <ProgramInputData>
                  <_ACTIVITY>Get Credit Check Result</_ACTIVITY>
                  <_PROCESS>CreditRequest#123</_PROCESS>
                  <_PROCESS_MODEL>CreditRequest</_PROCESS_MODEL>
                  <ControlDataObject>
                        <CorrelationID>XYAZ4711</CorrelationID>
                        <Event>RiskFactor</Event>
                  </ControlDataObject>
            </ProgramInputData>
      </ActivityImplInvoke>
  <WfMessage>
```

Fire Event Activity

Business processes are executed on a process engine.

◆◆◆

Specific states of business processes must be communicated to some unknown target systems and/or functions of systems unknown to the business process need to be initiated by a process activity.

A business process in execution (a business process instance) is a component with its own data space. However, the process stands in logical relation with other components, i.e. systems outside the process engine or other process instances. During the execution of business processes sometimes states are generated that need to be communicated to the space outside a process instance, e.g. to inform other systems about the completion of certain business activities. Moreover, sometimes the execution of functions needs to be initiated by a process but the process has no knowledge what component will execute that function. In both cases the process has no knowledge about the target system.

For this reason, this situation cannot simply be solved by invoking a service of a target system, as the target system that offers the service is not known. The process does now know how the function is going to be fulfilled and by what system. For instance, in case the process needs to communicate a state it might even be that the state is relevant to many other systems and not only to one system. Still the process does not know what systems require this information and the consequences created from that state in these systems. Furthermore it might be that the constellation of target systems is subject to regular change or that the systems are outside the influential space of the organisation hosting the process instance, e.g. in case the systems are placed in a restricted security area or are hosted by external organisations.

In all these cases, the business process must actually not know anything about the target systems and the business process design must not be dependent on them. As a result the business process must simply be able to publish some state information or to initiate functions by a process activity without knowing what target systems will take care of it.

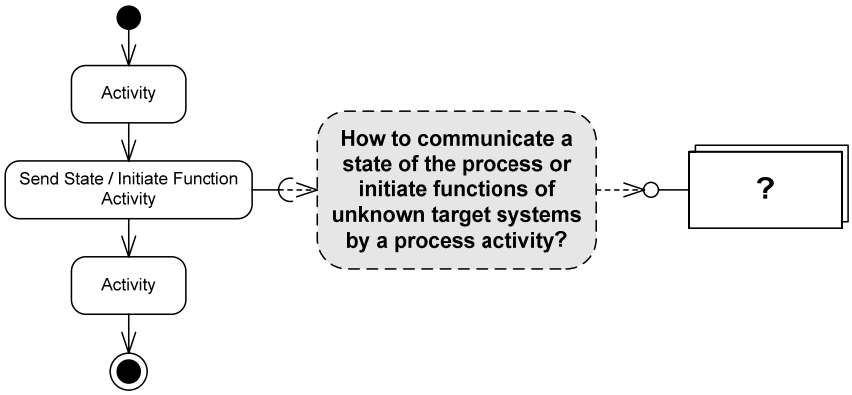

Figure 23: How to communicate with unknown target systems?

◆◆◆

Interpret the states to be communicated and the initiation of external functions as events generated by process activities. Model FIRE EVENT ACTIVITIES that represent event sources. These activities fire appropriate events, when they are reached in the process flow. Target systems subscribe to the events as event listeners and are responsible for processing the events.

Each process activity must have an implementation, i.e. some component that realizes the function of the activity. In the case of a FIRE EVENT ACTIVITY, the activity implementation represents an event source. Like any other process activity, the FIRE EVENT ACTIVITY delegates the execution of the function associated to it to its implementation. The function associated to a FIRE EVENT ACTIVITY is to create events. More precisely, its function is to create the specific events associated to a specific FIRE EVENT ACTIVITY in a business process. For this reason, different FIRE EVENT ACTIVITIES in a business process may create different events, as they potentially represent different event sources.

An event source notifies event listeners that have registered themselves to the event source about occurring events. The event listeners process the events they are notified about. An event listener is thus an OBSERVER [Gamma et al. 1994] of an event source. Any possible target system may implement an event listener. That way, the target systems are responsible to process the events, but the business process does not need to know them explicitly or have knowledge about what they do with the events. From a business point of view, logically some contract might be associated to an event, i.e. there may be specific requirements associated to the event on how to process it. If such requirements exist, the observer of the event is also responsible for considering them. However, whether such requirements exist or not depends on the specific business context.

How the FIRE EVENT ACTIVITY delegates the execution of its function to its implementation may vary. Generally, any technology offered to invoke a function by a process activity can be used. Figure 24 illustrates the a structure of a FIRE EVENT ACTIVITY.

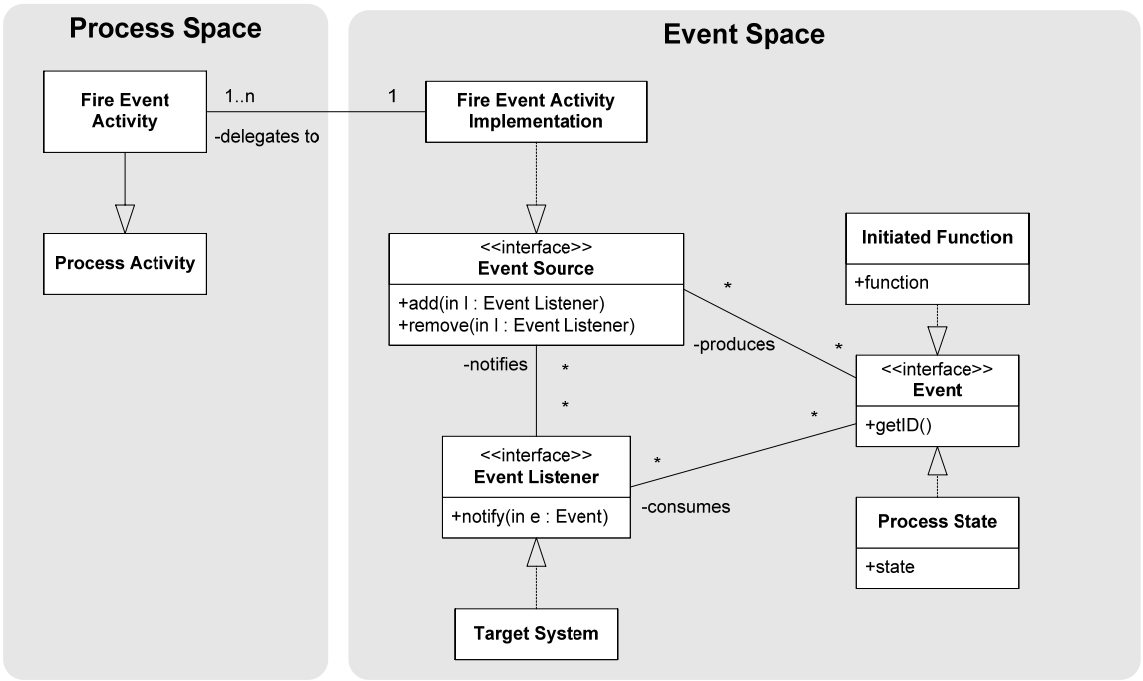

Figure 24: Structure of a fire event activity

A business process is able to communicate with unknown target systems by applying this pattern. The target systems may change without affecting the business process design or implementation. Another business process may also be a target system. In that way the pattern can be used to allow basically independent business process instances to communicate with each other. The pattern implies a functional architecture that represents additional implementation effort.

The invocation of the function realized by the implementation of the process activity that fires the event can be designed as a SYNCHRONOUS SERVICE ACTIVITY or a FIRE AND FORGET SERVICE ACTIVITY. If the events fired by a process activity have implications on other business processes that wait for these events to occur, then the EVENT-BASED ACTIVITY pattern applies [Köllmann et al. 2006].

Some known uses of the pattern are:

- In principle, the pattern is supported by the process technologies given as known uses in the SYNCHRONOUS SERVICE ACTIVITY pattern as basic support for synchronous services is required. Provided that this support is given the pattern can be implemented.
- In real project situations it often appears that processes are interrelated. For instance, in order management processes as we find them in the telecoms industry, there are usually parts of larger orders being processed in separate business processes. At a certain point in time these partial orders need to be consolidated. In order to achieve this each partial order process implements FIRE EVENT ACTIVITIES to communicate that a partial order has passed a certain state, e.g. the partial order is fulfilled. A coordinating process collects all the events and coordinates the progress of the overall order.
- In the insurance business we find similar scenarios in claims handling. Larger claims contain sub-claims and the processing of some complex claims takes up to several years. Each sub-claim thus runs as a single process and the overall claim also needs to be coordinated. This is also achieved by implementing FIRE EVENT ACTIVITIES in the sub-claims processes to achieve loose coupling of the interrelated claims processes.
- IBM's WebSphere Process Server offers direct technology support of the pattern by offering event generation and even event handling features in its tooling. BEA Fuego also offers such event generation and event handling features.

Example

A typical example is a process that cancels a complex order, e.g. in a telecoms company. Consider that an order may contain various products being ordered, especially if it is a business customer that places the order. Each product might have its own ordering business process and fulfilment process which is further orchestrated by some umbrella business process that coordinates the overall order. What happens if the customer cancels the whole order, while the order is still in fulfilment? In this case, it does not make sense to go on with the order fulfilment. But how do the order fulfilment processes of each single requested product recognize that the whole order has been cancelled?

The cancellation is a separate business process, which is independent from the ongoing order processes. In this case the cancellation process does also not know whether there are ongoing ordering processes. It might be that the fulfilment processes have also already executed business activities in external organisations, e.g. a cable provider, by invoking services of these external partners. Thus, the cancellation of the order should result in rolling back these business activities and also stop all ongoing order fulfilment business processes. As the business processes do

basically develop independently and should be loosely coupled, the FIRE EVENT ACTIVITY pattern is used to publish the cancellation event. Any partner system or business process interested in this event can capture it and react correspondingly.

The order fulfilment business processes contain EVENT-BASED ACTIVITIES [Köllmann et al. 2006] that implement event listeners, react on this event and stop the fulfilment processes in a controlled way. Another event listener informs external business partners about the cancellation. This way the cancellation process is loosely coupled with the external world. However, the business logic, caused by the raised event, can still be executed without the cancellation process knowing about any system involved in this event. Each system that listens to the event is responsible for processing the event (separation of concerns). The cancellation process is not responsible for it, as it is not able to decide on all the consequences that may result out of the cancellation event. As a result, the complexity generated out of this parallelism becomes manageable by applying the FIRE EVENT ACTIVITY pattern.

Asynchronous Sub-Process Service

Business processes are modelled with sub-processes and are executed on a process engine.

◆◆◆

From a logical business perspective one can observe two types of sub-process relationships. The first is basically like a functional de-composition and the second is more like an asynchronous invocation where the calling process does not wait until the sub-process has finished execution. Unfortunately, this second variant is usually not directly supported by process engines. It is thus an issue to depict this business requirement without having direct conceptual support from the process engine.

A sub-process is just an encapsulation of business activities that have some value in terms of reusability or that represent a coherent complex logical unit of work. When invoking a sub-process in another process, the normal behaviour of a process engine is defined as follows: step into the sub-process, step through all activities of the sub-process, and then return to the invoking process and continue with the remaining activities after the sub-process invocation. Hence, sub-processes are basically a functional de-composition. In some cases this behaviour is not wanted. It is required that the calling process should not wait until the sub-process has finished its execution, but instead it should directly continue with its own activities after sub-process invocation.

When business modellers create business process models they often implicitly assume that kind of asynchronous sub-process relationship. It is rather like initiating another process than having it fully enclosed in the calling process. As a result, when depicting these business processes on a process engine and more formal models of the processes need to be created, it is actually an issue how to model this asynchronous sub-process relationship, if the process modelling constructs of the process engine do not support it.

◆◆◆

Model an activity that invokes an ASYNCHRONOUS SUB-PROCESS SERVICE. This service only functionally encapsulates the instantiation of the sub-process on the process engine but not the whole execution of the sub-process.

Instead of modelling a sub-process via the process modelling language of the process engine, a SYNCHRONOUS SERVICE ACTIVITY or a FIRE AND FORGET SERVICE ACTIVITY is modelled that invokes a service that just instantiates the desired business process. The name/identifier of the process to be instantiated and the input parameters are provided as inputs to the ASYNCHRONOUS SUB-PROCESS SERVICE by the activity. For this reason, the control data object that is provided as input to the SYNCHRONOUS SERVICE ACTIVITY or the FIRE AND FORGET SERVICE ACTIVITY must contain attributes to represent these parameters.

The only result that the ASYNCHRONOUS SUB-PROCESS SERVICE might provide is an acknowledgement that to the invoking process that indicates whether the creation was successful or not. The service wraps the API of the process engine and creates the process instance via the API. A structure of an ASYNCHRONOUS SUB-PROCESS SERVICE is shown in Figure 25.

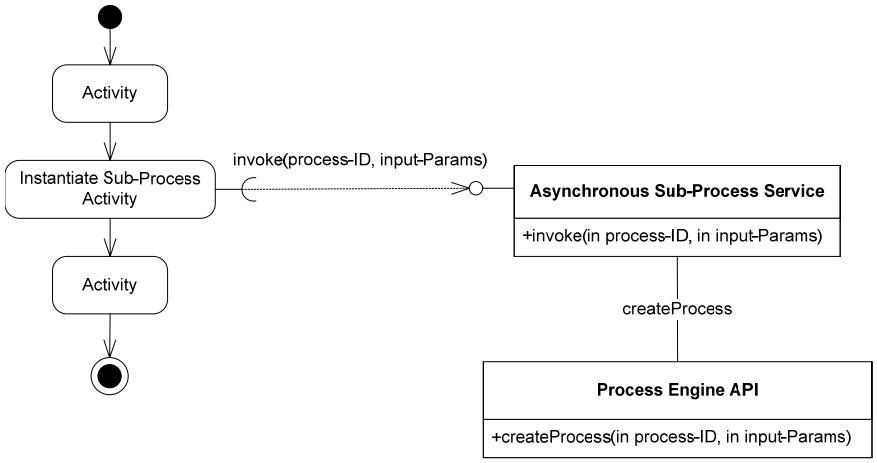

Figure 25: Structure of an asynchronous sub-process service

Asynchronous instantiation of sub-processes is possible without direct support of the process modelling language of the process engine. The process engine must have an API that allows to creating a process instance of a defined process with defined input parameters.

Apart from the SYNCHRONOUS SERVICE ACTIVITY and the FIRE AND FORGET SERVICE ACTIVITY patterns to realize the ASYNCHRONOUS SUB-PROCESS SERVICE that have been mentioned already, there are some other patterns related to this pattern. In order to provide a more loosely coupled relationship between the service and the API of the process engine or to provide more flexible ways of instantiating processes, the pattern can also be realised using the FIRE EVENT ACTIVITY pattern and the EVENT-BASED PROCESS ADAPTER [Köllmann et al. 2006].

The FIRE EVENT ACTIVITY is used to fire an event that represents the request for instantiation of a process. The EVENT-BASED PROCESS ADAPTER is an event listener that picks up the event and creates the requested process instance. This way another level of functional flexibility is provided, as the service request and the process instantiation are more loosely coupled. As a result, both these patterns in combination represent a conceptual functional architecture pattern for ASYNCHRONOUS SUB-PROCESSES.

Some known uses of the pattern are:

- In a large programme in an insurance company in the UK the pattern has been applied to offer a service provided on a service bus to instantiate asynchronous processes. This service can be called flexibly by business processes to instantiate various processes asynchronously. FileNet P8 Business Process Manager has been used as a process engine. That way it has been possible to invoke another FileNet workflow from a FileNet workflow asynchronously. As a result, the pattern has become an essential part of the architectural standard of the programme. WebSphere Business Integration Message Broker has been used for implementation of the service bus and service has been offered as a Web service.
- In a larger project in the automobile business the pattern has been used to offer asynchronous instantiation of processes based on WebSphere MQ Workflow. The service has been offered as an MQ Series based messaging interface. The pattern has been applied based on the same technology and principles in other projects such as telecoms and banking. Actually, these MQ based implementations can be found even before SOA has become a broader known term. There are other implementations that use the same principles in other industries such as banking and telecoms. The same pattern applies

when modern Web services technology is used.

Example

WebSphere MQ Workflow provides a Java API that can be used to instantiate a process via an ASYNCHRONOUS SUB-PROCESS SERVICE. The service can be invoked with WebSphere MQ Workflow just according to the example in the SYNCHRONOUS SERVICE ACTIVITY pattern. Thus, the process-ID and the input parameters are provided in an XML message. The service itself can be realized as a Java program that listens to the queue, picks up the XML message, instantiates the process via the Java API of MQ Workflow, and sends a return message based on the result of the instantiation. The Java implementation to create the process instance is quite simple.

```
import com.ibm.workflow.api.ExecutionService;
import com.ibm.workflow.api.FmcException;
import com.ibm.workflow.api.ProcessTemplate;

import java.util.Vector;

private ExecutionService service;


...

final class WorkflowSessionMQWF
{

   ...

   public void createProcess(String processName,
                             ProcessInputDataMQWF inputData)
                             throws WorkflowExceptionMQWF
   {
      ProcessTemplate[] template = null;
      try {
            String temp = "NAME='"+processName+"'";
            template = this.service.queryProcessTemplates(temp, "NAME",
                           new Integer(1));
            if(template.length == 0){
                  throw new WorkflowExceptionMQWF(
                             WorkflowExceptionMQWF.NO_TEMPLATE_EXCEPTION);
            }
            if(inputData == null){
                  throw new WorkflowExceptionMQWF(
                             WorkflowExceptionMQWF.NO_INPUTDATA_EXCEPTION);
            }
            template[0].createAndStartInstance2(processName, "", "",
                             inputData.getContainer(), false);
      } catch (FmcException e) {
            throw new WorkflowExceptionMQWF(e);
      }
   }

   ...

}
```

The implementation shows basically an application of the API of MQ Workflow to instantiate a process. The Java code shows that the process is identified by a unique name, representing the process-ID as defined in the pattern. The process name is given to the *createProcess* method as an input parameter. Other input parameters are passed in an object of type *ProcessInputData*. This

data represents the actual input parameters of the process.

First the appropriate process definition is queried as identified by the process name. This is done invoking the *queryProcessTemplates* method of the *ExecutionService* class, which is part of the API. If there is no process definition (process template) found for this name then an exception is thrown. After that, an instance of the process definition is created by calling the *createAndStartInstance2* method of the *ProcessTemplate* class.

Terminable Delivery Service

Business processes are executed on a process engine.

◆◆◆

Often the flow logic of a business process is dependent on certain states of business objects. The business process expects certain conditions related to business objects to be present by some defined deadline—these conditions reflect the state of the business objects. The process logic thus needs to distinguish whether the business objects related conditions are met in time or not.

Business objects are manipulated via business processes. The business objects are created, updated, or deleted. Often business objects, or rather a certain state of business objects, influence the control flow of business processes. That means, the business processes are dependent on certain conditions related to the business objects. For this reason, business objects are used for synchronizing the execution of business processes.

Consider a business process that creates a customer business object and another business process that is executed in parallel and which may only proceed beyond a certain activity if the customer business object has been created. That way the activities of different business processes are coordinated by the common customer business object. Usually, the coordination is also time bound, i.e. the conditions associated to those business objects have deadlines, representing the latest point in time when the condition must be true in order for the dependent process to proceed normally.

◆◆◆

Model a TERMINABLE DELIVERY SERVICE that is invoked by a SYNCHRONOUS SERVICE ACTIVITY. The service checks a desired condition with a given deadline of a defined business object. It delivers a result that indicates whether the condition is true or false and whether the deadline is reached or not. If the condition is false and the deadline is not reached, then invoke the service again by modelling a loop in the business process.

The TERMINABLE DELIVERY SERVICE takes a business object ID, a condition that specifies the desired state of the business object, and deadline for the delivery of the state as input parameters. The output of the service provides information whether the condition is true or false for the required business object and whether the deadline is passed or not. The business objects needs to be available in a CENTRAL BUSINESS OBJECT POOL [Hentrich-1 et al. 2006]. The service retrieves the business object identified by its ID from the CENTRAL BUSINESS OBJECT POOL and checks whether the condition is true for the business object and whether the deadline is reached. The service reports both results of the checks back to the invoking activity.

The process model distinguishes three business events. First, the condition is true and the deadline is not reached. Second, the condition is false and the deadline is passed. Third, the condition is false but the deadline is not reached, which leads to another invocation of the service after some time has elapsed to check again. Logically, a fourth case is also possible which is that the deadline is passed and the condition is true. However, the time intervals should be designed that way as to avoid this fourth case, as is actually not a valid business event. Figure 26 shows a structures of a TERMINABLE DELIVERY SERVICE .

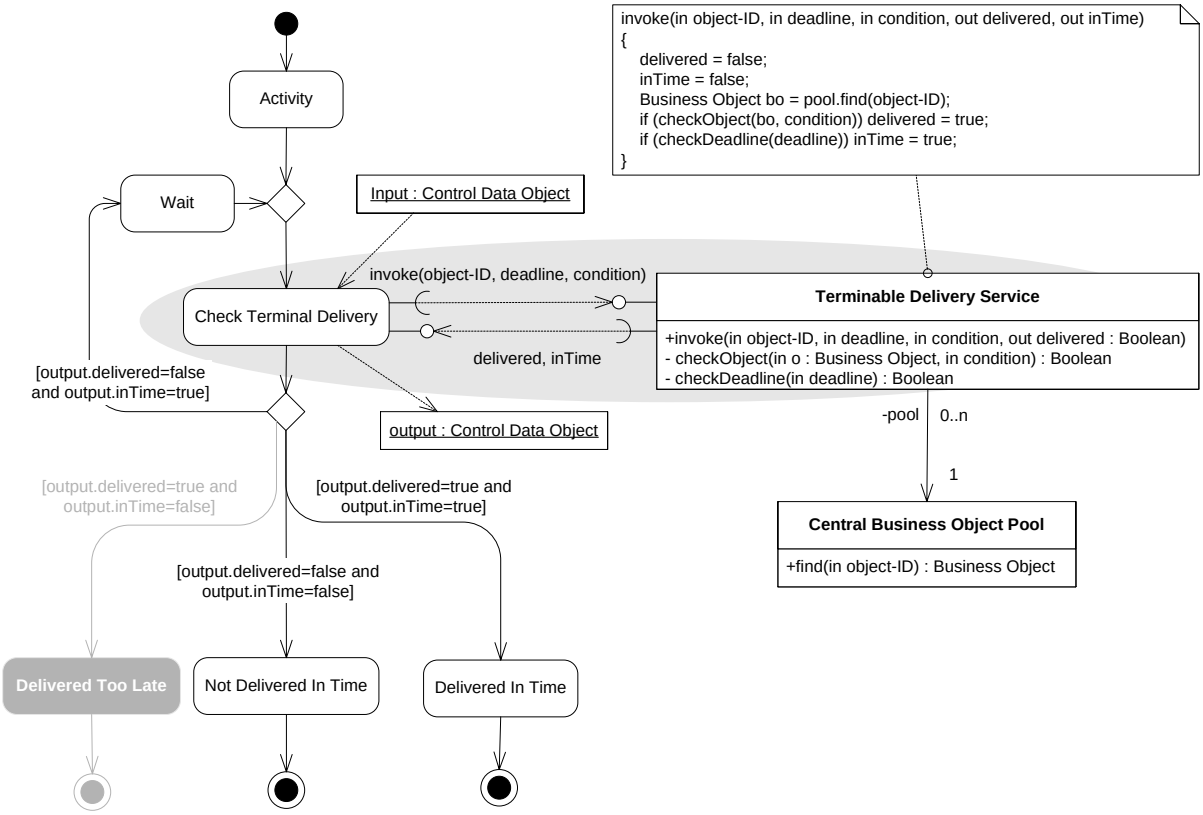

Figure 26: Structure of the terminable delivery service

It might be hard to design one generic service that checks all possible conditions for all types of business objects. In reality often multiple dedicated services need to be designed that check for fixed conditions on a certain type of business object associated to the service.

Some known uses of the pattern are:

- IBM WebSphere Process Server offers off-the-shelf features to query for certain states in a database. The Information Aggregator component currently allows doing inline SQL queries from a process. This way it is possible to query for a certain state and to react in the process accordingly. BPEL offers looping functions to re-run the query if the deadline is not exceeded.
- The BEA tool Fuego which is based on BEA Aqualogic offers very similar database features to easily implement the pattern. This is another example of a broader technology support of the pattern.
- In Enterprise Content Management processes in the insurance and telecommunications business the pattern has been used to implement waiting positions in processes for documents that have been requested. The forthcoming example will illustrate this in more detail. Similar implementations can also be found in banking, for instance. Many of the document based processes show implementations of the pattern.

Example

An example of a business requirement for this pattern is the expected arrival of documents. One can imagine a business process that processes a customer order. At a certain stage during the

order fulfilment process, the signed contract of the customer is necessary. The signature of a contract is usually time-bound. The customer is asked to send the signed contract back within 14 days, for instance. If the signed contract does not arrive within this time interval, then the order fulfilment process must not proceed normally. Thus, the decision logic of the business process is dependent on the event of the document arrival in conjunction with a certain deadline. The document must not only just arrive it must also be properly signed by the customer. This reflects the problem that there are certain time limits associated to desired states of business objects and the decision logic of the business process must consider this somehow.

Thus, if the signed contract arrives this will imply a change on a business object. The state of the business object will change as to reflect the arrival of the documents. The documents might be stored as associated objects to the customer object. The service will thus check for the appropriate state, which might simply be indicated by a corresponding status attribute of the customer object, or rather some aggregated business objects as the customer might have several open orders. The object ID provided to the service references the open order and the service checks for the status. If the status indicates that the contract associated to the order has been signed, the service will report this information accordingly to the process.

Conclusion

In this paper we have introduced a small pattern language for service integration in process-driven and service-oriented architectures. The patterns capture various types of service invocation: synchronous, fire and forget, and asynchronous invocation with one or multiple replies. The patterns illustrate how these types of service invocation need to be reflected in process models in order to integrate processes with services. Moreover, functional architecture implications are also captured by the patterns. That means the patterns do not only just deal with the integration of services and processes but they also deal with functional architectural design issues of the services.

The patterns reflect solutions for general business requirements that can be found in SOA engagements. The language presented in this paper is thus another building block in developing a comprehensive pattern language for process-driven and service oriented architectures, which we started to develop in our previous papers [Hentrich 2004, Hentrich-1 et al 2006, Hentrich-2 et al. 2006, Köllmann et al. 2006, Zdun et al. 2006].

References

[Barry 2003] D. K. Barry. Web Services and Service-oriented Architectures, Morgan Kaufmann Publishers, 2003

[Channabasavaiah 2003 et al.] K. Channabasavaiah, K. Holley, and E.M. Tuggle. Migrating to Service-oriented architecture – part 1, http://www-106.ibm.com/developerworks/webservices/library/ws-migratesoa/, IBM developerWorks, 2003

[Evans 2004] E. Evans. Domain-Driven Design – Tackling Complexity in the Heart of Software", Addison-Wesley, 2004.

[Gamma et al. 1994] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[GFT 2007] GFT. GFT Inspire Business Process Management. http://www.gft.com/gft_international/en/gft_international/Leistungen_Produkte/Software/Business_Process_Managementsoftware.html, 2007.

[Hentrich 2004] C. Hentrich. Six patterns for process-driven architectures. In Proceedings of the 9th Conference on Pattern Languages of Programs (EuroPLoP 2004), 2004.

[Hentrich-1 et al. 2006] C. Hentrich, U. Zdun. Patterns for Business Object Model Integration in Process-Driven and Service-Oriented Architectures, Conference on Pattern Languages of Programs (PLoP), Portland, Oregon, 2006.

[Hentrich-2 et al. 2006] C. Hentrich, U. Zdun. Patterns for Process-Oriented Integration in Service-Oriented Architectures, In Proceedings of the 11th Conference on Pattern Languages of Programs, (EuroPLoP 2006), 2006.

[Hohpe et al. 2003] G. Hohpe and B. Woolf. Enterprise Integration Patterns. Addison-Wesley, 2003.

[Köllmann et al. 2006] T. Köllmann, C. Hentrich. Synchronization Patterns for Process-Driven and Service-Oriented Architectures, In Proceedings of the 11th Conference on Pattern Languages of Programs, (EuroPLoP 2006), 2006.

[Voelter et al. 2004] M. Voelter, M. Kircher, and U. Zdun. Remoting Patterns - Foundations of Enterprise, Internet, and Realtime Distributed Object Middleware. Wiley Series in Software Design Patterns. J. Wiley & Sons, October 2004.

[Zdun et al. 2006] U. Zdun, C. Hentrich, and W.M.P. van der Aalst. A Survey of Patterns for Service-Oriented Architectures, Internet Protocol Technology, Inderscience, 2006.

Appendix: Overview of Referenced Related Patterns

There are several important related patterns referenced in this paper, which are described in other papers, as indicated by the corresponding references in the text. Table 2 gives an overview of thumbnails of these patterns in order to provide a brief introduction to them for the reader. For detailed descriptions of these patterns please refer to the referenced articles.

Pattern	*Problem*	*Solution*	*Category*
BUSINESS OBJECT REFERENCE [Hentrich 2004]	How can management of business objects be achieved in a business process, as far as concurrent access and changes to these business objects is concerned?	Only store references to business objects in the process control data structure and keep the actual business objects in an external container.	Architecture
BUSINESS-DRIVEN SERVICE [Hentrich-2 et al. 2006]	How can the requirements be engineered to decide what services need to be defined, what functionality is actually required, and thus what services must be designed and implemented?	Design BUSINESS-DRIVEN SERVICES that are defined according to a convergent top-down and bottom-up engineering approach, where high-level business goals are mapped to to-be macroflow business process models that fulfil these high-level business goals and where more fine grained business goals are mapped to activities within these processes.	Functional Architecture
CENTRAL BUSINESS OBJECT POOL [Hentrich-1 et al. 2006]	Business processes are very often interdependent in their flow logic, such that a running process may generate circumstances that have effects on other processes being executed in parallel.	Keep the business objects in a central pool such that they can be accessed in parallel by all processes of the process domain.	Technical Architecture

Pattern	*Problem*	*Solution*	*Category*
CORRELATION IDENTIFIER [Hohpe et al. 2003]	How does a requestor that has received a response know to which original request the response is referring?	Each response message should contain a CORRELATION IDENTIFIER, a unique identifier that indicates which request message this response is for.	Technical Architecture
EVENT-BASED ACTIVITY [Köllmann et al. 2006]	How can events that occur outside the space of a process instance be handled in the process flow?	Model an event-based activity that waits for events to occur and that terminates if they do so.	Event Synchronization
EVENT-BASED PROCESS ADAPTER [Köllmann et al. 2006]	How can process instances be created on a process engine on the basis of occurring events?	Use an event-based process adapter that instantiates processes if corresponding events occur.	Event Synchronization
GENERIC PROCESS CONTROL STRUCTURE [Hentrich 2004]	How can data inconsistencies be avoided in long running process instances in the context of dynamic sub-process instantiation?	Use a generic process control data structure that is only subject to semantic change but not structural change.	Interface
MACROFLOW ENGINE [Hentrich-2 et al. 2006]	How is it possible to flexibly configure macroflows in a dynamic environment where business process changes are regular practice, in order to reduce implementation time and effort of these business process changes, as far as the related IT issues are concerned that are involved in these changes?	Delegate the macroflow aspects of the business process definition and execution to a dedicated MACROFLOW ENGINE that allows developers to configure business processes by flexibly orchestrating execution of macroflow activities and the related business functions.	Technical Architecture
MACROFLOW INTEGRATION SERVICE [Hentrich-2 et al. 2006]	How can the functionality and implementation of process activities at the macroflow level be decoupled from the process logic that orchestrates them, in order to achieve flexibility, as far as the design and implementation of these automatic functions are concerned?	The automatic functions required by macroflow activities from external systems are designed and exposed as dedicated MACROFLOW INTEGRATION SERVICE with well-defined service interfaces.	Functional Architecture
MICROFLOW ENGINE [Hentrich-2 et al. 2006]	How is it possible to flexibly configure IT systems integration processes in a dynamic environment, where IT process changes are regular practice, in order to reduce implementation time and effort?	Delegate the microflow aspects of the business process definition and execution to a dedicated MICROFLOW ENGINE that allows developers to configure microflows by flexibly orchestrating execution of microflow activities and the related BUSINESS-DRIVEN SERVICES.	Technical Architecture

Pattern	*Problem*	*Solution*	*Category*
PROCESS BASED ERROR MANAGEMENT [Hentrich 2004]	How can errors that are reported by integrated applications in activities in a process flow be handled and managed?	Define special fields for error handling in the process control data structure and embed an activity in an error handling control flow.	Process Modelling
PROCESS INTEGRATION ADAPTER [Hentrich-2 et al. 2006]	How can interface and technology specifics of a process engine be connected to a different interface and technology of another system, such that the two systems can communicate, as far as requests for activity execution and the corresponding responses are concerned? How to design this connection in a loosely coupled fashion?	Use a PROCESS INTEGRATION ADAPTER that connects the specific interface and technology of the process engine to an integrated system.	Technical Architecture
PROCESS-BASED INTEGRATION ARCHITECTURE [Hentrich-2 et al. 2006]	What architecture design concepts for process-driven backend systems integration are necessary, in order for the architecture to be scalable, flexible, and maintainable?	Provide a multi-layered PROCESS-BASED INTEGRATION ARCHITECTURE to connect macroflow business processes and the backend systems that need to be used in those macroflows.	Technical Architecture
REPOSITORY [Evans 2004]	Exposure of technical infrastructure and database access mechanisms complicates the client.	Delegate all object storage and access to a REPOSITORY.	Technical Architecture
TIMEOUT HANDLER [Köllmann et al. 2006]	How can timeouts of process activities be managed in a process?	Model a timeout handler that defines behavior in the process model in case a timeout has occurred.	Process Control Flow Synchronization
WRAP SERVICES AS ACTIVITY [Hentrich-1 et al. 2006]	Existing interfaces of external systems often do not reflect the requirements of a process-oriented SOA. Loose coupling – a main goal of any SOA – for instance is often not well supported because the external system only offers stateful interfaces.	For each domain entity in the external systems define one stateless SERVICE on top of the existing interfaces of the external system. A special SERVICE activity type is defined in the process engine that wraps invocations to external services.	Functional Architecture

Table 2: Thumbnails of referenced patterns