Exception handling in web service processes

Dietmar Jannach and Alexander Gut

Technische Universität Dortmund,
44221 Dortmund, Germany
{firstname.lastname}@tu-dortmund.de

Abstract. Cross-company business processes are common in today's networked economy and are nowadays often supported by process support systems that integrate the information systems of the different partners based on web service technology. In contrast to earlier Workflow Management Systems, which were often deployed in the controlled environment of a single company, the distributed nature of modern solutions make company-spanning web service processes more susceptible to failures. Therefore, it is desirable to augment the process models already at design time with error-handling behavior such that disrupted process instances can for instance be rolled back or completed on an alternative execution path, if, e.g., an individual service is not reachable.
In this chapter, we will give an overview of past and current approaches as well as potential future works to exception handling in web service processes. We start with the concepts that were developed in the area of Workflow Management Systems, continue with error-handling techniques in state-of-the-art process modeling languages and finally give an outlook on future automated approaches to error recovery and repair.

1 Introduction

In the last quarter-century, many companies have successfully (re-)organized their business according to the *process-oriented paradigm* – as opposed to product- or function-oriented models – in order to increase efficiency and withstand competitive pressure. Alongside with this evolution, information systems that support this process-oriented way of organizing a business have been developed and are now well-established in industrial practice. Beside large and integrated Enterprise Resource Planning (ERP) systems, which comprise a wide range of cross-domain configurable processes, and specialized domain-specific solutions, *Workflow Management Systems (WfMS)* came up in the early 1990s as a means to support electronic business processes and accompanying document flows.

One particularity of this type of *Process Support Systems* is that such systems are based on *process models*, which are not pre-implemented but can be designed according to the needs of the business. Thus, most Workflow Management Systems consist of a design-time component, which is used to graphically sketch the process flow (also called *workflow schema*), and a run-time component called workflow *engine*, whose main task is to execute the process flow by e.g., forwarding work items to the appropriate users.

A main assumption of very early WfMSs was that the supported processes do not go beyond company borders. Although other WfMS-external information systems or software solutions may be integrated, the engines were primarily designed to run in a "controlled environment" within the company's organizational and IT infrastructure. Likewise, the problem of dealing with unexpected situations during process execution was not in the focus of the software tools and/or research in the early years. However, the need for a defined way of dealing with "broken" workflows is obvious as in reality, things not always work out as planned. In the real world, typical ways of dealing with problems during a process comprise, e.g., the cancelation or undoing of already performed steps or the execution of alternative, compensating actions.

Early commercial WfMSs had no special, built-in means to *model* what should happen in case of an unexpected situation or upon a failure of an external tool. In these systems, the only option is to extend the process model (workflow schema) with explicit error-checking conditions and execution branches for *expected* problems. Typical measures to forward recovery of problems anticipated by the process designer might include re-doing a step (n times), canceling the process, skipping, retrying or postponing an activity and so on. However, even for very small processes, such an approach is not viable in practice because the resulting models become too complex when an additional execution branch is defined for every activity whose execution may fail. In fact, adding error recovery activities to the model in order to introduce fault tolerant behavior may paradoxically even lead to a *reduced* fault tolerance due to increased system complexity [HA00]. Therefore, researchers soon began to work on more elaborate approaches to augment the workflow models with additional error-handling constructs and engines with corresponding error recovery capabilities. The first proposed approaches were based on the concepts of transactions known from the field of database management systems, where the problem consists of ensuring a consistent system state even if one or the other step in the process fails, see [SR93] for an early work.

Later on, in the late 1990s, companies became more and more connected based on Internet technology and emerging e-Business standards. Consequently, the problem of unexpected behavior during workflow execution became more and more important as workflows could now span different companies and integrate various autonomous information systems, which are out of the control of the main driving workflow engine[1]. Note that from a technological perspective, application integration was a hot issue then, when commercial implementations of the CORBA[2] specification reached some level of maturity. Beside transactions, at that time also the concept of exception handling – inspired by the exception handling constructs of object-oriented programming languages – was introduced to workflow models as a means to error recovery, see [HA00] for an example.

[1] In fact, interface definitions for the integration of different workflow engines had already been included in the Workflow Reference Model developed by the Workflow Management Coalition around 1995; http://www.wfmc.org/

[2] http://www.omg.org/corba/

Today, in the late 2000s, the term "workflow management" is out of fashion. Cross-organizational electronic business process support, however, is more important than ever in today's networked economy. Thus, the problem of unexpected errors has remained and only the technologies have changed. Instead of CORBA, XML-based document formats and web service technology are used today. Instead of proprietary notations of WfMS vendors, (various) standardized process modeling languages such as WS-BPEL[3] are common for what is called "orchestrated web services" or web service processes.

In this chapter we will first give an overview of how fault tolerant behavior, based e.g., on exception handling or transaction concepts, can be expressed in process models in state-of-the-art modeling languages. The considered modeling languages are the Business Process Modeling Language BPMN[4], WS-BPEL, and YAWL[Hof05]. With respect to transactions and co-operating web services, we will also shortly discuss the corresponding "choreography" web service standards.

In the last sections, we will summarize current research efforts toward so-called "self-healing" web services based on extended process models and sketch how semantic annotations for services can serve as a basis for next-generation, automated recovery mechanisms.

2 Process modeling languages

Over the last two decades, several proposals for languages for business process modeling have been proposed by individual research groups or by industry consortia. While we could observe some convergence of the competing proposals in the last years, e.g., in the form of language mappings, there exists no "'Lingua Franca" today that is consistently used across all different communities that have to do with business process modeling. In the following, we will discuss which error-handling mechanisms are supported by three popular languages. The languages are the (a) BPMN, which is based on informal semantics and graphical models only, (b) YAWL, as a language from academia with its origins in workflow modeling and (c) WS-BPEL, as today's de-facto standard for modeling orchestrated web service processes.

2.1 BPMN

In 2002, the Business Process Management Initiative (BPMI) developed the Business Process Modeling Notation (BPMN) as a new graphical notation for modeling business processes. One of the particular design goals of the BPMN was that it should be easy to understand for all involved stakeholders, i.e., both for the business and the IT people. Later on, in 2005, the BPMI merged with the Object Management Group (OMG), an influential standardization body which also maintains the Unified Modeling Standard UML. Similar to UML, the BPMN

[3] http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html
[4] http://www.omg.org/spec/BPMN/, current version is 1.2

specification describes the semantics of the graphical symbols that can be used in the diagrams only in an informal way. Note that in the past, approaches that are based on informal semantics have been quite successful and are wide-spread in industry despite the problems that are caused by missing formal semantics.

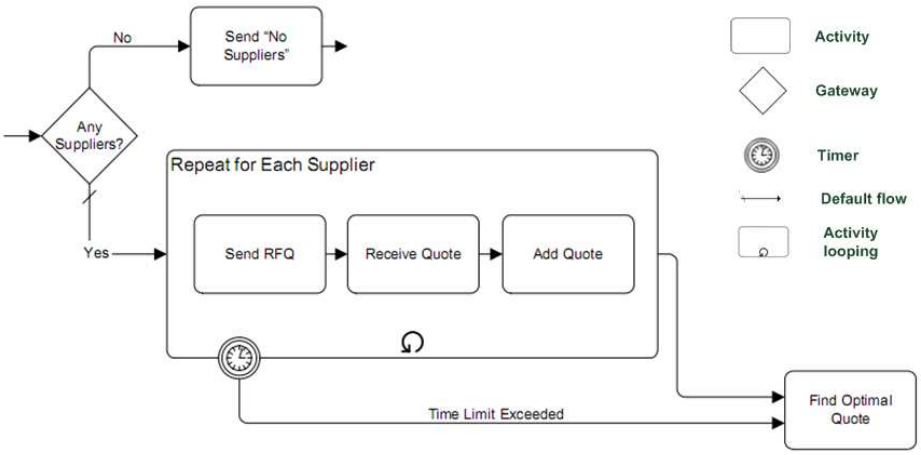

Fig. 1. Process example from BPMN specification

Figure 1 shows a simple example process model from the BPMN 1.1. specification. The main modeling elements of the notation are activities (tasks), sequences, so-called gateways (for modeling decisions or parallel execution), loops and events such as incoming messages or – as in the example – a timer event.

In particular, with the help of gateways and events, alternative execution paths for expected problems can be – at the price of increased model complexity – quite easily modeled in BPMN. However, BPMN also includes additional notational elements to model error handling behavior.

A so-called "Exception Flow" can be seen in Figure 1, where the normal flow in the loop is interrupted when some time limit is exceeded. Note that although this is called an Exception Flow in BPMN, in this case it corresponds more to a construct for loop-termination or timeout treatment than to exception handling in programming languages.

Figure 2, also taken from the specification, shows a process model that includes more built-in error handling modeling elements of BPMN. Error handling is based on transactions (modeled as sub-processes), error and exception events as well as on compensation actions. The intended business logic of the example from the travel planning domain is as follows. The process foresees that the buyer will only be charged, when both the flight and the hotel reservations are successful. If one of the bookings fails, the transaction should be rolled back by executing "undo" (cancel) actions for already completed tasks. In the model, each action within the transaction is therefore associated with a compensat-

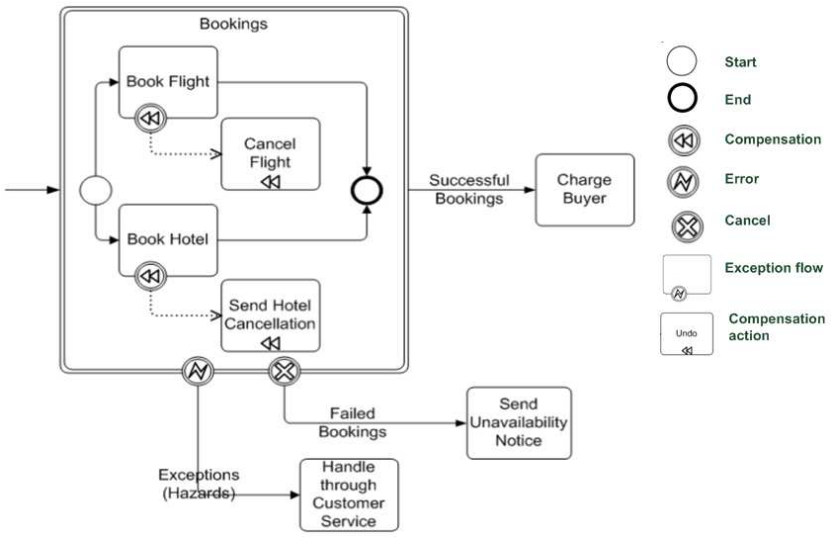

Fig. 2. Error handling example from BPMN specification

ing task (marked with the *fast rewind* symbol). The scope of the transaction is graphically defined through the double lined border. The start of a rollback process is initiated when a "Cancel" event happens, which can be triggered by a "Cancel End Event" within the transaction or, as in this example, when a cancel message is received from the underlying transaction layer (e.g., WS-Transaction[5] or the Business Transaction Protocol[6]). If such an event happens, a "Cancel Intermediate Event" (cross in a circle) directs the execution flow to the activity for sending an unavailability notice. This action is executed when all cancelation actions have been completed. The process model in Figure 2 also contains an "Error Intermediate Event" (lightning in circle) for situations, in which "something went terribly wrong". In this case, no compensation is initiated and the flow is directly diverted to the error handling activity.

Beside these intuitive pre-defined and dedicated error-handling mechanisms, the concept of "ad-hoc" processes can in principle also be used to provide for flexible error handling in case of an exception. An ad-hoc process is defined as a group of activities that have no pre-defined execution order, i.e., only the actual performers (determined at run time) decide on the execution flow. Thus, one can design some pre-defined ways of reacting on problems and delay the decision of what to do until the problem actually occurs. Since processes are typically long-running and often require human interaction, this mechanism can be used to incorporate more flexibility in the recovery phase. This mechanism might be particularly helpful when it would be too complex to model all possible problem

[5] http://www.oasis-open.org/committees/ws-tx
[6] http://www.oasis-open.org/committees/business-transaction/

causes and corresponding reactions, when some information is not available in electronic form, or when human decision taking is required.

Discussion. The error-handling concepts are very general and built upon ideas that have already been developed in the context of WfMS, see e.g., [EL97] or [EL96], who propose sub-processes as transactions, backward recovery and undoing of already performed steps. Workflow Recovery in the sense of a partial backtrack and continuation of another execution path [EL96] can also be specified with BPMN with the above-described concepts.

Remember that BPMN is only a graphical notation that has no formal semantics and its design is based on the idea that also business people can design the process models. In that context one might ask whether it is realistic that business people are capable of understanding the exception handling concepts. For IT people, transactions, events and handler routines are known concepts but for business analysts, extensive training might be necessary to use the concepts (if the used drawing tool supports them anyway). For these reasons, syntactically incorrect BPMN diagrams can often be found in practice. If the enhancement of processes with error handling behavior is thus left to the IT department, a part of the value of a language that can be used by both sides is lost. In that context, the fact that BPMN has no formal semantics and can only be translated or mapped to other representations such as WS-BPEL or XPDL[7] can lead to additional problems when tool vendors interpret the mapping itself or the details of transactions, events and intended error-handling behavior in different ways.

A recent approach toward achieving compatible error-handling behavior across different products and overcoming the problem that fault-tolerant models soon become very complex is described by Combi et al. in [CDP08]. Their work is based on modeling so-called *triggers* with the high-level Chimera-Exception language [CCPP99] to define the exception handling rules. These triggers are then automatically mapped to XPDL definitions, i.e., a compiler produces *enriched* XPDL models from the annotated process models that for instance contain an additional exception handling lane. Although Combi et al. report some improvements with respect to portability of the process models, they also mention that most of today's commercial WfMSs do not fully comply to the XPDL standard yet or rely on proprietary language extensions, which hampers cross-product portability, in particular of fault-tolerant process models.

2.2 Yet another workflow language - YAWL

One of the main issues of notations such as the BPMN is that they have no precisely defined formal semantics, which easily leads to different interpretations of the same standard by the tool vendors. One intuitive way of modeling the control flow of a workflow process with precise formal semantics are Petri Nets. Since their development in the 1960s, the basic scheme has been extended with time-related concepts and with distinguishable tokens that carry data (Colored Petri Nets). However, as argued in [Hof05], even such *high-level Petri nets* have

[7] A BPMN serialization format by the WfMC, see http://www.wfmc.org/xpdl.html

some limitations when it comes to workflow modeling. One of the problems is related to the *locality of events*, i.e., tokens are only enabled when their directly connected input places are loaded. On the other hand, they can also only affect their direct successors. This aspect of locality is problematic, in particular when a *cancelation* procedure has to be executed in case of an external event such as a timeout: When no other modeling constructs are available, one has to model a so-called "vacuum cleaner" to remove tokens from different places in the web, which in turn leads again to complex models.

As an answer to this and other problems of high-level Petri nets, ter Hofstede proposes a different formalism called YAWL[Hof05]. The design of this language is based on the analysis of the functionality of various system and the formulation of commonly occurring *workflow patterns*. YAWL formalism itself uses *extended workflow nets*, which consist of tasks (activities) and conditions (similar to Petri Net places). The workflow nets can also be structured hierarchically.

In the context of error handling, the *cancelation pattern* corresponds to the situation where an (external) event should cause the cancelation of individual running or scheduled activity or of the whole process.

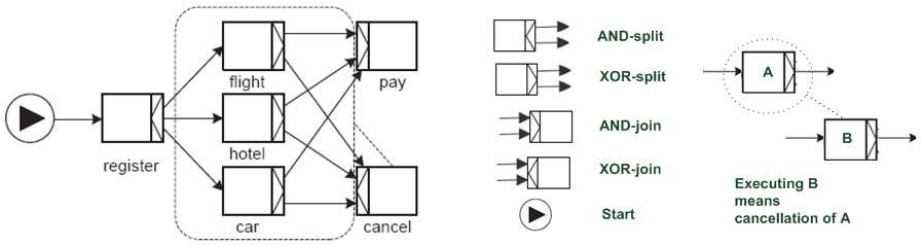

Fig. 3. YAWL Cancel activity pattern

Figure 3 shows the "Cancel Activity/Region" pattern from [Hof05]. YAWL notation has to be read as follows: When the `cancel` task is executed, which happens when one of the booking fails, all tokes within the region surrounded by the dashed lane (cancelation region) are removed, i.e., the other booking tasks are also cancelled. With the help of this pattern, complex constructs for token-removal in different parts of the graph can be avoided. The cancelation pattern can be combined with the event-handling mechanism of YAWL to model *expected* exceptions. Events are simply modeled as special tasks: timed tasks (denoted with a T) execute automatically after a certain waiting time, event tasks (denoted with an E) wait for an incoming (external) event.

Figure 4 (also from [Hof05]) shows how events and timers can be combined with cancelation. In that example, a time out will cancel the waiting for the payment. Vice versa, upon receipt of a payment, the timer will be canceled.

While with the help of these constructs *expected* problems can explicitly be modeled with decision tasks, they cannot be used to model exceptions in the

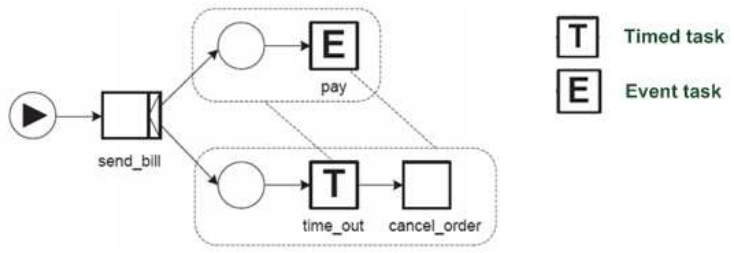

Fig. 4. YAWL Cancel activity with events and timer

sense of programming languages, where the control flow is diverted automatically to some error-handling activities as for example in BPMN.

Therefore, Russel et al. in [RvdAtH06] extended their library of workflow patternswith a set of additional *exception handling* patterns and corresponding notational elements for YAWL. The following general types of exceptions were identified: *Work Item Failure* (work item cannot progress for some reason), *Deadline Expiry*, *Resource Unavailability*, *External Trigger* (also from other process instances), and *Constraint Violation* (based on explicitly modeled invariants).

The exception patterns are based on a combination of three considerations:

- What is the state of the work item in the work item life cycle?
- How will already started workflow instances be handled? Options are continuing the instances or removing the current or all work items.
- Which remedial actions are taken? Options are rollback, compensation, or "no-action".

Based on this characterization of exceptions, the authors propose to introduce additional symbols to the notation and in order to model so-called *exception handling strategies*. These strategies imply some sort of control-flows that contain both error-handling tasks as well as standard YAWL activities. The main point however is that the exception handling behavior is modeled separately (in an own sub-diagram) and only linked to the normal control flow. In addition, different exception handling flows can be designed that depend on the type of the exception (i.e., a work-item failure may require other actions than a deadline expiry) or even on different data values of the current work item. Based on this separation, not only the mixture of business logic and error handling is avoided, it also allows for easier verification and maintenance of the resulting models.

A different method for exception handling called "Exlets" was proposed in the context of YAWL by Adams et al. in [AtHvdAE07]. They base their method on the concept of *Worklets* [AtHEvdA06], which was devised as a new approach to introduce additional flexibility in the workflow models. Worklets are small workflow processes, which are designed to handle specific (sub-)tasks in the context of another process. The main idea is that the "main" workflow process is not seen as a rigid prescription of the series of actions to be completed but rather as a

guideline. Depending on the current situation during workflow execution, an individual (atomic) task of the main process can be substituted by a Worklet, i.e., the process control is handed over to the small Worklet process. The selection of the appropriate Worklet is driven by a decision engine that evaluates contextual selection rules, which are modeled as *"Ripple Down Rules"*[CJ90]. One particular advantage of the method is that the set of available Worklets (i.e., the repertoire of possible actions) can be extended even at run-time allowing for incremental workflow evolution.

Conceptually, "Exlets" are very similar to "Worklets" with respect to their basic nature, i.e., they are small work processes that are dynamically invoked during the execution of some *parent* process and their selection is driven by decision rules[8]. The main difference is that – depending on the workflow model – the Worklet selection process is triggered only by specific tasks, while checks for Exlet execution are run for every *case*. Technically, the so-called *Exception Service*[AtHvdAE07] can be implemented as a separate (service-oriented) component and be invoked from different workflow execution engines using defined interfaces for data mapping. If the Exlet invokes another Worklet (that e.g., cancels or undoes other tasks), this Worklet can again be monitored by the Exception Service.

Although there are no limitations with respect to what can be modeled in an Exlet, it is proposed that an exception engine should also support the above mentioned Exception Patterns[RvdAtH06]. The system described in [AtHvdAE07], which was implemented in the YAWL-environment, supports seven of ten defined exception types: constraints that are checked before and after item and case execution, timeouts and externally triggered exceptions. Other mechanisms, which depend stronger on the internals of the workflow engine such as item abort, resource unavailability, or events that occur during work item execution, were not implemented in the system.

Discussion. The original YAWL method only supported the *cancelation Pattern* as a way of dealing with problematic situations during process execution. Later on, *Exception Patterns* were proposed to enhance YAWL in this context, mainly on the conceptual level. Only the last addition to YAWL based on the Worklet extension provides comprehensive error-handling support, which allows for easy modeling of re-do activities and alternative execution paths. In addition, a further main advantage of the Worklet-based exception modeling technique in YAWL is that the normal flow and the "exception flow" can be designed and evolved – even at run-time of a case – in separate models. On the other hand, due to the loose coupling, ensuring that the exception handling behavior is correct and works as intended, may become more complex.

2.3 WS-BPEL

Today, the *Web Services Business Process Execution Language*, short WS-BPEL, is the de-facto standard for modeling "executable" workflow models that are

[8] The Exlet process may again invoke a Worklet process with compensation tasks.

based on web services. The language (called BPEL in the following) supersedes two earlier languages for *programming in the large*: The Web Service Flow Language (WSFL) by IBM and Microsoft's XLANG and was submitted to OASIS for standardization by IBM, Microsoft, SAP, Siebel and BEA in 2003. The current version (WS-BPEL 2.0) was released by OASIS in 2007.

In BPEL, the flow of activities in a business process – which are implemented as web services – are described with the help of an XML format since BPEL itself has no graphical notation. Therefore a BPEL document contains several elements such as the list of activities and the corresponding control flow elements (if-else etc.), definitions of code blocks (called *scopes*) and variables that store instance data. Beside these elements, which can also be found in many programming languages, incoming messages and (asynchronous) events and the corresponding handler definitions play an important role in BPEL. Since BPEL was designed as a language for orchestrating (distributed) web services that can be provided by different partners, the language design naturally includes different mechanisms for handling errors and developing fault-tolerant applications.

The basic error handling mechanism in BPEL are *Fault Handlers*. Conceptually, BPEL fault handlers are similar to exception handling blocks in programming languages, i.e., the code in the fault handling block is executed in case of an error and the block can also *re-throw* an exception so that it can be handled in an outer block. In addition, different fault handlers can be defined for different types of problems and can be attached to the various (nested) parts of the process: error-handling routines can be defined for a single activity, a *scope*, or the process itself.

Note that in contrast to normal, single-threaded programs, parallel execution of different tasks is not uncommon in web service processes. Before the error-handling procedure begins, all running activities of the affected scope have to be terminated first, which is implicitly done by a fault handler. Optionally, a *Termination Handler* can be used for some cleanup activities.

Beside graceful termination and notification, the cancelation of already performed activities is another core element of the error-handling features of BPEL. To this purpose, the language supports the definition of explicit *Compensation Handlers*, which contain a list of activities to be executed to undo things. Thus, when an explicit *compensate* activity is invoked in the fault handler, the compensation handler for the affected scope and already performed activities will be executed.

Figure 5 shows an example of BPEL's fault handling mechanism[9]: The sales business process consists of two activities, which are organized in own scopes: charging the customer's credit card and (electronic) shipment of the goods. Let us assume that a problem occurred in the second step. The *ship goods* activity fails and control is handed to the local fault handler attached to the scope. This fault handler propagates (re-throws) the error to the surrounding scope, which happens to be already the main process. This fault handler first executes a

[9] Adapted from König and Baretto's WS-BPEL symposium presentation (2007), www.oasis-open.org/events/symposium/2007/slides/WS-BPELWorkshop.ppt

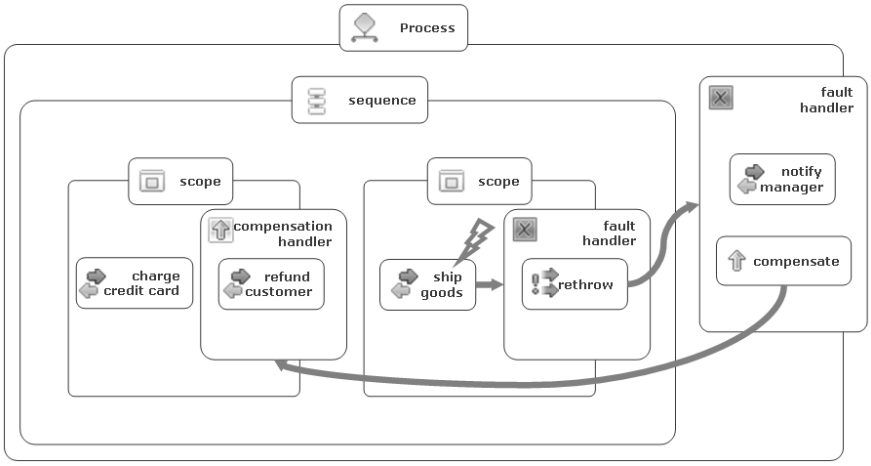

Fig. 5. BPEL fault handling and compensation activities

notification activity and than initiates the compensation process, i.e., it invokes the compensation handler of the already executed activity *charge credit card*, which contains the compensating task *refund customer*.

Beside the fault handler, also the event processing mechanism of BPEL can on principle be used to incorporate fault tolerant behavior in the process models. BPEL supports two types of events. *Alarm events* are raised when a user-defined timer goes off, i.e., they can be used to model *expected* timeout exceptions. *Message events*, on the other hand, are triggered by inbound messages. They can for instance be used to capture failure notifications of a connected system.

Discussion. BPEL basically supports the *catch-and-handle* approach to model error-handling behavior and follows the principle of nested exception blocks and compensation handlers. Based on these principles, a reaction also to unexpected problems is possible and not all types of possible problems have to be explicitly modeled within the normal flow.

What cannot easily be expressed in BPEL is the strategy of *ignoring*, but logging problems. In some situations, some errors should be caught and someone should be notified, but the process should continue as planned. Once the control is handed to the fault handler in BPEL, however, all running activities are terminated. The only exception in BPEL is the *suppressJoinFailure* setting, which allows to quietly ignore false join conditions. Also the "redo" scenario is not well-supported in BPEL. Although it is of course possible to redo activities in a scope by copying the corresponding part of the process into the compensation handler, this leads to an undesired duplication of the process logic and increased model complexity.

Because of these limitations of BPEL with respect to error handling, Modafferi et al. in [MMP06] propose to extend BPEL engines with "self-healing" mechanisms for error recovery (SH-BPEL). Their approach is based on pre-processing

the model and using additional annotations in the process models from which standard-compliant, but more fault-tolerant, BPEL-definitions can be generated that support re-doing of a scope, rollback to a defined (save)point in the process, or the choice of an alternative execution path.

Finally note that that BPEL has no graphical notation. This fact may be problematic when stakeholders from the business side should *model* the intended error-handling behavior of the process, since the XML-based format and the underlying language principles are not easy to comprehend by non IT-experts.

2.4 Other "programming in the large" modeling languages

In industrial practice, many other (graphical) languages are in use for modeling business processes. However, many of them are designed for special purposes such as business process analysis, simulation and system design and will not be discussed in detail here, since they either are proprietary, have no defined formal semantics or are not actually designed for web service process execution.

Examples are the ADONIS standard language (discussed for instance in [Gla08]) or Event-Driven Process Chains [STA05], notations which are used for process analysis and simulation purposes and do not have special constructs for exception handling. In some cases, UML activity diagrams are also used to model business processes. In recent versions, UML does support the concept of exception handling and introduces an own graphical symbol[10]. While this is a feature that is probably inspired by BPEL, it is the only error-handling mechanism in UML Activity Diagrams known to the authors. Moreover, the missing formal semantics of UML leaves room for interpretations. Despite this fact, in [GM06], a first proposal was made to transform Activity Diagrams into executable XPDL documents. However, the above-mentioned exception handling notation is not supported in this approach. Finally, a notation for modeling exception handing in UML Sequence Diagrams was proposed in [HH06] but is not yet contained in the standards.

3 Choreography Models

Modern computerized business processes often need to integrate the services of different business partners in a single workflow, e.g., on the basis of web service and BPEL technology. The coordination model of BPEL is called "orchestration", i.e., there is one partner that owns and steers the process and whose workflow engine invokes the partner systems.

The other service co-ordination model is that of "choreography". In this model, all services are equal (i.e., there is no dedicated coordinator) and only some distinguished aspects of a service are visible to the others. A *choreography* model therefore only contains these external service descriptions as well as agreements between them that may relate to sequencing, timed behavior, or transactional aspects.

[10] OMG UML 2.1 Superstructure Specification, p. 361

Early approaches toward web service choreography were based on *Behavioral Interfaces*, which describe the observable behavior of *one* single service in a choreography. The *Web Service Choreography Interface (WSCI)*[11] is an example of such a language for describing the externally visible behavior of a web service. This XML-based language was submitted to W3C already in 2002. The proposal reached the status of a *note* and was not developed further. Later on, however, the main concepts of choreography were integrated into the *Web Service Choreography Description Language* (WS-CDL)[12].

The design of WSCI demands that the observable behavior of services is defined declaratively through the description of the service interface and the temporal and logical constraints on the interaction with other services. The descriptions themselves are not executable and it is left to the web service designer how the behavior is actually implemented.

Technically, WSCI is based on an extension of the standard Web Service Description Language (WSDL)[13]. While WSDL can only be used to describe the different messages of a service (including their technical characteristics), a WSCI document can further extend these definitions, e.g., with information about how newly arriving messages are related with previous messages.

Beside message dependencies, WSCI also supports the description of the error-handling behavior of a particular service. Possible errors are, e.g., WSDL error messages, messages with wrong format, timeouts and errors raised by the service itself. The following listing illustrates the timeout behavior of an online payment service and specifies that when the allowed time for entering the identification number has passed, a particular compensation service has to be executed in order to undo the transaction.

```
<exception>
   <onTimeout property = "tns:PINEntryTime"
              type= "duration" reference="tns:preparePayment@end">
      <compensate transaction = "tns:cancelPayment"/>
   </onTimeout>
</exception>
```

As can be seen in the example, WSCI supports similar error-handling concepts as "programming in the large" approaches such as BPEL, i.e., events, transactions and compensation. The difference is that no global view and coordination exists and the descriptions are only defining the behavior of a single service (e.g., which compensating actions are invoked in case of an exception) in a declarative manner.

In contrast to WSCI, which only describes the behavioral interface of a *single* service, WS-CDL aims to extend this model and support a global (but not centrally coordinated) view on the choreography. Since 2005, WS-CDL has the status of a *candidate recommendation* at W3C.

[11] http://www.w3.org/TR/wsci/
[12] http://www.w3.org/TR/ws-cdl-10/
[13] http://www.w3.org/TR/wsdl

A description of a choreography in WS-CDL has in some respect a similar structure as XPDL or BPEL process definitions. It does not only describe the roles and relationships of the different partners in the process, but also the different activities that the partners can execute along with their associated control flow structures in the form of *WorkUnits*. This concept of WorkUnits can also be used to describe the error-handling behavior in the choreography. Exceptions that occur at run-time can be caught by so-called *ExceptionBlocks* and be handled in *exception workunits*.

Discussion. Although WS-CDL can be used to describe similar aspects of a process as BPEL, BPMN or XPDL, the goal of the WS-CDL designers is to have descriptions that are even more independent from the actual implementation below. For example, the individual actions do not have to be web services and could also be Java or C# programs or executable BPEL processes.

While this independence from technical details is desirable in particular from an academic perspective, choreography approaches suffer from their limited acceptance in practical environments. Even the standardization efforts at W3C have come to a halt in the last years, probably because BPEL in the current version contains the concept of *abstract processes*, which supports the description of interfaces without requiring a specification of the concrete execution logic. Recently, approaches toward a further enhancement of BPEL (BPEL4Chor) with features required for choreography modeling have been proposed in [DKLW07]. Since BPEL is today's de-facto standard for modeling web service processes, we expect that languages such as WS-CDL will be replaced in the future by choreography-enhanced process modeling languages.

4 Toward more intelligent error handling methods

The approaches discussed so far are mostly based on the throw-and-catch error handling pattern as well as on transaction rollbacks or compensation actions, which are explicitly modeled by the designer of the process. In practical applications, such approaches have their limitations in particular when it comes to the problem of recovery or repair, i.e., when the problem is to complete the process despite the presence of an error. If, for instance, one of the involved web services is not reachable, in many domains the option exists to use another service that provides the same functionality; another option in that situation could be to try a different set of activities to reach the same goal (think, e.g., of different payment services).

With current approaches it is possible to model such recovery actions only through explicit alternative execution paths or compensation actions. Since, in reality, many or all activities of a process may fail and every single failure may require different recovery activities, the resulting process models soon become too complex. As a result, process designers will only model repair actions for a few of the possible error cases and will accept a rollback for the other situations.

In the following, we will summarize recent approaches toward more intelligent error-handling techniques that support the precise identification of the problem,

the automated discovery of alternative web services, or the dynamic computation of suitable recovery plans.

4.1 Error diagnosis and repair

In order to intelligently handle a problem during process execution, e.g., through the usage of an alternative service that provides the same functionality, one goal could be to find out in a first step, which of the activities could have caused the problem. Only at the first glance, this looks like an easy problem, which can be solved by modeling an exception handler for every single task in the process. Note, however, that a service can also raise an exception due to wrong input data calculated by an erroneous preceding activity. Thus, the goal of *diagnosis* techniques is to identify a set of already executed activities that could have potentially contributed to the observed problem.

Most recent approaches toward advanced error analysis for workflows are based on *Model-based Diagnosis*(MBD) [Rei87] techniques. Figure 6 shows the basic principle of Model-Based Diagnosis, which was originally developed to diagnose physical systems such as electronic circuits but later on also used to debug software systems, see e.g., [FSW99], [Wot02], or [FFJS04].

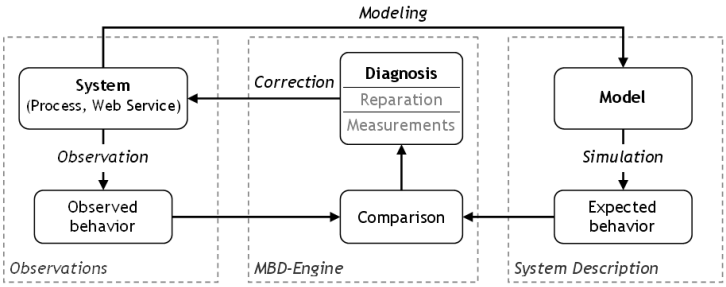

Fig. 6. Principle of Model-based Diagnosis

As the name suggests, the approach is based on a model of the real-world system, which is often referred to as SD, the *System Description*. In an electronic circuit diagnosis scenario, SD contains the list of elements of the circuit (e.g., the different gates), their interconnections and also a description of the relevant physical laws. Based on this model, the system behavior can be simulated. In the domain of electronic circuits, we would for example test different inputs to the circuit and observe how the values are propagated to the output lines.

Basically, the diagnosis approach consists of a comparison of the *observed* behavior of the real system with the *expected* behavior, which we can determine with the help of a simulation run. If there are discrepancies, we know that that there is some error in the real system (assuming that the model is correct and captures reality properly). The problem now is to find out, which of the

components of the system causes the error and should be replaced. In MBD-literature, a set of components of SD that – if assumed to be faulty – explain the observed behavior of the system, is called a *diagnosis*. Typically, we are only interested in *minimal* diagnoses, i.e., we do not want to include components in the diagnosis that are not relevant for explaining the error. The set of (minimal) diagnoses for a given problem can for instance be calculated with Reiter's *Hitting Set* algorithm[Rei87].

Figure 6 also indicates that finding a diagnosis is normally not the ultimate purpose of model-based problem analysis. Commonly, one is interested in repairing the system, e.g., by replacing individual components. Note that in typical problem settings there are usually several diagnosis candidates that can explain the observed faulty behavior of a process. The (automated) selection of the most probable or appropriate one for a given situation is an open and critical issue in many MBD-based approaches, because also the corresponding repair actions can be different for different diagnoses.

Diagnosing BPEL process through dependency analysis. In [YD07], Yan and Dague propose a method for monitoring and diagnosing web service processes based on the MBD paradigm. In order to apply MBD techniques, a formal model of the web service process has to be developed. Therefore, the existing process definition (given in the BPEL4WS language) is first transformed into a Discrete Event System, that is, into a *synchronized automaton*, where the states are defined by the values of the variables of the process. State transitions are caused by variable changes that are in turn triggered by *actions* such as variable assignments, message emissions or receptions. The model of non-abnormal behavior for every activity in usual MBD notation is:

$$\neg ab(A) \wedge \neg ab(A.inputs) \Rightarrow \neg ab(A.outputs) \tag{1}$$

The non-abnormal (correct) model states that if activity A behaves correctly (i.e., is not *abnormal*) and the inputs to A are also correct, then also the outputs of activity A, i.e., the resulting variables changes, are correct.

Once an exception occurs during run-time execution, the diagnosis phase is initiated and the process starts comparing the *observed behavior* (given by the execution logs of the engine and the corresponding execution "trajectory") with the *expected behavior* (as described by the automata and the given variable dependencies).[14]

At run-time, the diagnosis engine is called when an exception occurs. The subsequent diagnostic process is based on three pillars.

– *Dependency analysis*: Similar to Program Slicing [Wei81], a software debugging technique, a (static) analysis is made of how variable values can be

[14] Note that to that purpose the BPEL engine must provide additional logging support and e.g., record incoming and outgoing SOAP messages as well as events and exceptions.

affected by activities. To that purpose, activities are viewed as functions that accept input variables (defined as global BPEL variables) and produce output variables. These relationships can be derived from BPEL's *correlation sets*, which describe how arriving messages are related, and are used to model the non-abnormal behavior of an activity more precisely, i.e.

$$\neg ab(A.input) \Rightarrow \neg ab(A.output), \text{if } Util(A, A.input, A.output) \quad (2)$$

The function *Util(A, V1, V2)* denotes that through activity A output variable $V2$ is correlated with the input variable $V1$.

– *Trajectory reconstruction*: The execution trajectories are recovered from the BPEL logs by synchronizing the observations (events and exceptions in the log) with the system description, i.e., the formal execution model. Figure 7 shows a simple process model with parallel activity execution (left hand side) and two execution trajectories. One corresponds to a possible successful execution path; in the other, an exception is thrown before the process finishes.

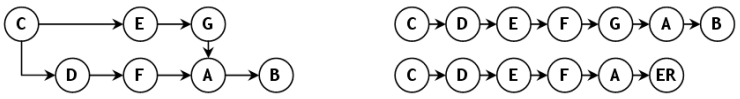

Fig. 7. Model and two execution trajectories

– *Accountability analysis*: In this step, the possible "culprits" (i.e., activities that may have caused the problem) of the exception are determined. The set of candidates is limited to those activities that have been already executed and to those who possibly have affected the input variables of the failed activity (which can also be the only responsible activity). This calculation is based on applying *responsibility propagation rules* through the given trajectory. If two or more exceptions happen simultaneously, the analysis is made in parallel and the resulting diagnoses are then combined. In case of sequential (chained) exceptions, only the first exception is analyzed.

Discussion. In principle, the approach is very similar to the well-known Program Slicing technique (with some slight differences). Thus, it applies a proven technique for ordinary programs to the "programming in the large" world. A deeper discussion of the relationship between Model-based Diagnosis for Software Debugging and Program Slicing can be found in [Wot02], who applies an even more elaborate diagnosis technique for Java programs.

One limitation of the approach by [YD07] is that the underlying BPEL engine must support the extended logging mechanisms, which are required to spot the problematic activity. In addition, the approach is also limited to certain types of failures. *Time-out* exceptions, for instance, which are probably caused by problems in a remote system, cannot be analyzed by the proposed technique.

Diagnosing workflow processes from "first principles" In order to overcome the problems of the approach from [YD07], Friedrich and Ivanchenko in [FI08a] propose a more fundamental approach to model-based debugging of workflow executions. Their problem formalization of the workflow semantics is not based on a specific language such as BPEL, but rather on implementation independent *Colored Petri Nets* (CPN) and they support common workflow control-flow patterns such as sequences, splits, choices, merges and joins. Loops (cycles) in the models are, however, not yet supported by their approach.

In order to properly model variable dependencies and process execution states, the authors extend CPNs with so-called (data) *objects* and call the models Workflow/Data-Petri-Nets (WFD). In addition to these data objects, the authors also make use of *guard* transitions (activities) that are similar to *assertions* in programming languages, i.e., at some given points during execution, certain Boolean expressions are checked. In the proposed formalism, a workflow exception therefore corresponds to a situation in which an assertion/guard expression evaluates to false. Together, all these mechanisms (data dependencies, guard transitions and the execution history of individual process instances) serve as a basis for identifying the potential sources of an exception more precisely in the WFD approach [FI08a].

In order to diagnose workflow executions from "first-principles", the extended CPN model (WF) and the execution behavior of workflows are expressed in a logical representation, i.e., by a set of First-Order Logic sentences. These descriptions serve as a basis for a precise formal and consistency-based characterization of diagnoses of workflow executions which can be calculated with the help of Reiter's Hitting Set algorithm [Rei87] and a theorem prover for consistency checking.

In a later work, Friedrich et al. [FMS10] reformulate the workflow diagnosis problem using on a different logic-based formalism and encoding. Their approach includes different improvements over the basic model-based debugging method and supports diagnosis-repair settings in which activities are potentially re-executed by the execution engine in order to successfully complete a process despite the existence of a fault. In contrast to dependency-based approaches, their method for identifying incorrect activities in process execution has the advantage that it does not require precise behavioral activities descriptions (which are often not available in loosely coupled systems based on web services).

Discussion. When compared with the work in [YD07], which represents more a Program Slicing than a Diagnosis technique, both share the idea of utilizing a formal model of workflow executions and the analysis of data dependencies. The approach of Friedrich and Ivanchenko aims to tackle the diagnosis problem in a more fundamental manner and to characterize diagnoses for workflow execution in general (and not dependent on dependency propagation). While there are some simplifying assumptions of their approach (e.g., loops are allowed in the model but can be dealt with using loop unfolding techniques), the authors show in [FMS10] that a variety of real-world problems can be modeled and solved with their formalism.

In addition to the mentioned approach of [YD07], the theory developed in [FI08a] also lays the foundation for the automated computation of possible repair actions for problematic workflow instances, see [FI08b] and [FFM+10].

Distributed diagnosis of web services. Yet another approach toward model-based workflow problem diagnosis that goes beyond today's standard throw-and-catch scheme, is proposed by Ardissono et al. in [ACG+05] and [AFG+06].

Their work is driven by the fact that web service processes often span several companies and that the individual elements of the global overall process can be seen as whole sub-processes that are executed at a partner system. Thus, a central point of problem analysis cannot easily be installed, because, e.g., the participating partners may not be willing to share all the internal information required for diagnosis to the coordinating partner. Therefore, the authors propose a model for distributed error handling as shown in Figure 8.

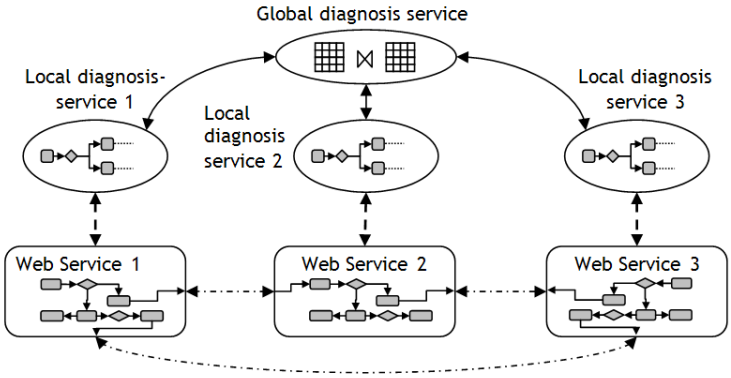

Fig. 8. Local and global diagnosers (adapted from [AFG+06])

It is assumed that every participant of the distributed web service process has a *local diagnoser* component, which uses the log information of the local web service to diagnose the reasons for the problem. Note however that a failure found in a service is possibly caused by an error in another service. Therefore, a *global diagnoser* is introduced, which can combine the information of the individual local diagnosers into a global hypothesis about the reasons of the problem; the communication itself is based again on web service technology and a defined message exchange protocol.

Overall, the main advantage of the approach is that the central diagnoser does not have to be aware of the internals of the other participating web services. Even more, the global component does not necessarily need to know in advance how the different services interact with each other.

Discussion. Similar to the other works, in the approach of Ardissono et al., activities are viewed as components of a system that transform inputs to outputs

and which can have the behavioral modes *ok* and *ab*(*normal*). In particular, every activity has three subcomponents that can fail as the authors distinguish between three types of variables: Variables, whose values should be simply forwarded, variables that are newly created through the activity and variables, whose values are changed by the component. In that respect, the model is a bit more detailed than the more simple input-output model of [YD07]. Details of how *activity execution chains* in the sub-processes are handled (e.g., based on trajectories as in [YD07]) are not given.

On the level of the infrastructure, the approach from Ardissono et al. is based on the assumption that every local service, which is for instance implemented as orchestrated BPEL process, communicates with the local diagnoser. For that purpose, the local web service engine has to be extended and – in case of BPEL processes – the fault handlers of the model have to be modified, so that the diagnoser is appropriately informed whenever a local exception occurs.

Overall, techniques from model-based diagnosis appear to be a promising way to deal with exceptions in web service processes, in particular as the basic mechanisms in many cases are to a large extent independent of the protocols and engines that are used in practical environments. What remains open in all discussed approaches is the question of computational costs for real-world web service processes. While it is known that the computation of diagnosis is in general a computationally complex problem, it would be interesting to know whether the proposed methods are applicable to process definitions of limited complexity that can be found in the real world.

Repair planning / contingency planning. Finding out which part of a web service process failed or causes a problem is helpful, when the goal is to exclude a certain service from being used in future process instances or when alternative services that provide the same functionality can dynamically be incorporated into the processes. In many cases, however, this repair scheme, which can be summarized as *"find an alternative service and re-try"* is not appropriate. It would therefore be desirable to have an agent who can determine dynamically whether a broken workflow instance can still be successfully completed, e.g., by executing a set of alternative activities. In other words, the agent needs to find a repair plan to complete the service or otherwise abort it.

In [FI08b], Friedrich and Ivanchenko propose such a method for computing repair plans based on the workflow formalism by [RD98], in which a workflow is represented by a directed acyclic graph of control nodes and activity nodes together with a set of associated *objects*. A process that terminates normally results in a certain configuration of the output objects. When an error occurs, the problem is to find a repair plan (which is itself a workflow) that leads to a configuration at the endpoint that is equal to the configuration that would appear after normal process execution.

The repair approach in [FI08b] is based on the usage of common repair actions such as (re-)execution (suitable, e.g., to deal with transient failures), activity substitution and compensation. In order to deal in particular with transient

failures, the concept of *time* has to be incorporated into the repair model. The starting point for repair planning is therefore a data record that describes which activity failed at which point in time[15].

Repair planning itself is based on a logical formalization of the workflow process model, the effects of the different repair actions, the time model and the workflow execution state. The authors of [FI08b] classify their repair planning procedure as "conditional planning with background theories".

The logical formalism was chosen in a way that it can be handled by the *dlv* disjunctive logic programming system [LPF+06]. How this system can be used for knowledge-based planning has first been shown by Eiter et al. in [EFL+04]. Later on an extension was developed to support (optimistic) conditional planning was later on developed by [vNEV07]. [FI08b] finally expand these ideas in a way guaranteeing that contingency plans always lead to successful workflow completions.

Discussion. An evaluation of various automatically generated test cases of small to medium sized workflow processes indicates that the method is generally applicable to real-world repair problems. The examined processes consisted of up to a few dozen activities. Computation times on a standard desktop PC remained less than a half-minute for most of the test cases, which is acceptable, taking into account that web service processes are typically longer lasting and have no tight time constraints attached. However, what has not been discussed are limitations of the approach, such as the problem of handling loops in the process models. It can be assumed that repair plans for such processes cannot be computed with that method yet (compare the work of the same authors in [FI08a]). Another aspect, which may constrain the applicability of the method in real world concerns the question whether one can assume that an in-depth description of the precise effects of individual activities (web services) is available in practical settings.

4.2 Semantic Web Technology

Replacing a transiently or permanently failing service of a process chain with an alternative service that provides the same functionality was already mentioned as a possible repair action in previous sections. Recent technology requires that the set of alternatives is known in advance and is described explicitly in the process model, e.g., in the exception handler of an individual action.

Still, it would be desirable that alternative services could automatically be discovered and used by an intelligent system in case of a problem. Consider, for instance, the problem of booking a travel arrangement automatically which involves various web services and activities such as flight or hotel reservations (compare [MSZ01]). If a particular online hotel booking service fails, the process should not be halted but the booking functionality of another e-tourism platform should be used instead. If we do not want to model the alternative services or

[15] Note that this information could be derived from a diagnosis step that precedes the repair phase.

the service discovery process explicitly, a more generic mechanism is required and the particular problem is to find a service (or a set of services) that provide the same – or at least a sufficiently similar – functionality as the failing service that caused the problem.

Today's standard technologies for describing web services such as WSDL allow us to model a service and its interface on the *syntactical* level, i.e., they can be used to list the set of known operations and their parameters. However, these technologies cannot be used to describe the *semantics* of the service, i.e., the *effects* of invoking the service. Therefore, in order to allow for automated discovery or – which is even more complex – the automatic composition of service chains to complete an otherwise failing process, mechanisms for precisely describing web service semantics are required.

In a broader context, the idea of annotating web resources in a way that they can be located intelligently or combined by a computerized system, is the core vision of the Semantic Web [BLHL01]. In recent years, significant research efforts went into the realization of this vision and lead to the development of a the Web Ontology Language OWL adopted by W3C[16].

Shared *ontologies* are the basis of the *Semantic Web* and define the set of concepts that can be used to annotate resources in the web, i.e., they help us to insure that the same terms and vocabulary is used and interoperability is guaranteed. OWL itself is a formal, XML-based and machine-processible language that can be used to describe specific domain ontologies.

An (OWL-based) ontology for the above mentioned travel-planning problem would for instance define the concepts "flight" or "hotel" and probably also describe that a "domestic flight" is a special subtype of a flight.

In the context of web service *discovery*, we could therefore try to extend the existing syntactic WSDL descriptions with semantic annotations, e.g., use the agreed-upon term "Amsterdam hotel booking" to denote that our service provides that functionality. Whenever a service fails in a process chain with this annotation, an alternative service with the same or a comparable annotation – according to a common ontology – could be invoked, if the parameter lists are somehow compatible.

If a simple replacement of a service is not sufficient and a different chain of service invocations is required to achieve the original goal, a different way of describing the web service semantics is required. Note that the computation of diagnoses and associated repair plans was also the goal of the approaches of [FI08b] or [YD07] described above. In these approaches, the required execution semantics were described in terms of the variable dependencies or the effects of activities on data objects. However, this representation mechanism is rather technically oriented and not well-suited for sharing execution semantics because modeling is done on the level of variables and their dependencies and not the level of a more abstract domain ontology.

A more general and implementation-independent way of capturing web service semantics is to view a service as an *action* in a STRIPS-style [FN95] planning

[16] http://www.w3.org/2004/OWL/

problem from the field of Artificial Intelligence (AI) and correspondingly model a service through its *inputs*, *outputs*, *preconditions*, and *effects* (IOPE), see also [MSZ01]. How such a technology can potentially help us to react on workflow exceptions more intelligently, will be described later on.

Web service discovery In this subsection we will first sketch the opportunities and open challenges of automated web service discovery, which – as mentioned above – can also be seen as a mechanism to finding alternatives for a transiently or permanently failing web service.

UDDI (Universal Description, Discovery and Integration) is a standard that can be used for publishing and locating standard WSDL-based web services in a company-wide or global registry and was originally designed with the concept of service-brokering in mind. Consequently, "matchmaking" between service providers and consumers was a central functionality that should be provided by such an exchange platform. However, the mechanisms provided by UDDI in that context are rather limited and mainly consist of API-functions for locating a service by name, through keyword-matching or category-based search. Beside names and categorizations, web service descriptions can only be given in plain text or keywords. Automatic matchmaking is therefore not easily possible as the information can be incomplete, misleading, contain ambiguous words and so forth. Even the use of classification schemes such as UNSPSC[17] may not be sufficient because different schemes may be used and these schemes are not compatible.

In [TAH07], Tsetos et al. summarize the key innovations and additional components that are required for *semantic* web service discovery and also propose different possible architectures. The concept of Service Annotation Ontologies (SAO), which can be used to precisely specify the service capabilities, is at the core of their considerations. One proposal for a standardized language for describing web service semantics in this form is OWL-S which was submitted to W3C for standardization in 2004. Note that competing proposals such as WSDL-S or WSMO[18] were submitted to W3C. In this chapter, we focus on OWL-S because for different reasons. First, OWL-S is designed in a way that the semantic annotations can contain concepts from domain-specific OWL-based ontologies. At the same time, OWL-S also allows us to "ground" an action and technically bind it to a standard WSDL- and SOAP-based web service.

According to the proposal of [TAH07], an extended web service discovery architecture can make use of the existing UDDI infrastructure, but the registry entries should have pointers to the unambiguous semantic descriptions expressed in the annotation language, i.e., the "Service Advertisements" should contain semantic descriptions. If the "Service Requests" are also stated in the same way, i.e., when the desired service capabilities are described in a semantically precise way, advanced *matchmaking* algorithms can be employed to decide whether a

[17] http://www.unspsc.org/

[18] http://www.w3.org/Submission/OWL-S/,http://www.w3.org/Submission/WSDL-S/, http://www.w3.org/Submission/WSMO/

given service matches the requirements. Since exact matches cannot always be expected, usually a measure for the "Degree of Match" (DoM) is introduced. Such a measure can be based on various aspects such as the overlap of concepts used in the advertisement and the request or the placement of the services in a classification hierarchy. Note that the DoM measure can not only be used for service discovery (in the sense of matching) but also for a subsequent ranking step, which is also common in such matchmaking architectures [KSF07].

Finally – besides sophisticated matchmaking algorithms – also modern ontology integration and mediation techniques can be utilized to align ontologies of different user communities; for an overview see, e.g., [ES07].

Technically, different architectures are possible. The semantic web service discovery system can either be centralized (e.g., as an extended UDDI service) or as a peer-to-peer system, in which every participant can act as a provider or requester of a service. For a centralized version, again different architectural choices are possible. In one proposal of [TAH07], the matching algorithms are themselves provided as services. Depending on the type of the request – keyword-based or semantics-based – the corresponding matching service and algorithm is selected dynamically.

Web service composition and re-planning Once semantic descriptions of individual services are available, it may not only be desirable to locate a single service but probably also to find a chain of services to reach a certain process goal or – corresponding to the focus of this chapter – find a *repair plan* to recover from a workflow exception. This process of deriving execution chains automatically is called web service composition and is often based on AI planning technology, see, e.g., [SPW$^+$04] or [HBP07]. Note that although the idea of deriving web service process chains automatically was the driving scenario in the beginning of Semantic Web Services research (see, e.g., the paper of McIlraith et al. in 2001 [MSZ01]), still no reports on practical business implementations can be found until today.

One system, in which Semantic Web Services and AI planning were combined successfully in the domain of server-side multimedia adaptation is reported in [JLTH06]. In this work, the goal was to design an open and extensible architecture for a multimedia server which has two basic features: First, it should be capable of integrating multimedia adaptation services and tools of different vendors and provides. Second, the server should be able to combine the different services into an invocation chain that converts a given resource (such as a video) from the given format to the desired format if a single-step adaptation is not possible.

In order to achieve these goals, the available adaptation operations (ranging from simple image resizing to complex content- and environment based video transformations) are modeled as Semantic Web Services (SWS) using OWL-S. An *action description* comprises the required in- and outputs as well as the preconditions for applying a transformation, i.e., the acceptable input formats, and the effects of the transformation that describe how the resource is changed after

applying the operator. Thus, given the set of actions, a "start state", which corresponds to a description of the resource to be transformed, and a "goal state", that corresponds to a description of how the end user wants to consume the resource, any standard state-space planning engine can be used to calculate a corresponding adaptation chain.

Note that in any domain in which Semantic Web Service technology should be exploited, the participants have to agree on a shared domain ontology determining, for example, how the different actions are described. The establishment of such an agreed-upon ontology often is a hampering factor when it comes to applications that should be open to a broad community of participants. In the domain of multimedia adaptation, this problem could be solved by interpreting the existing ISO/MPEG standards (MPEG-7, MPEG-21) as a domain ontology, as these standards exactly prescribe how the different features of a resource (such as color, size etc.) have to be described [JL07].

While the works described in [JLTH06] and [JL07] were not primarily designed for exception handling in web service processes, this example helps us to illustrate how SWS technology can serve as an enabler for more advanced error-recovery techniques. By design, the system can handle additions and removals of individual services. As long as there exists a possible execution chain that uses the currently available services, the system will find it due to the soundness and completeness of the underlying planning algorithms. If a service transiently or permanently is not available, alternative services can be retrieved automatically based on the semantic annotations of their effects.

Moreover, since the composition (and repair) problem is reduced to a well-understood AI planning problem, the known techniques for contingency planning [PC96] or re-planning can be applied, see e.g., [Pee05] for a replanning algorithm for web services. The most simple method would be to halt the execution chain when a problem is detected and generate a new plan that leads from the intermediate state to the original goal state. More elaborate replanning and rescheduling techniques that for instance aim to produce a new plan which has minimal deviations from the original plan are also possible, see [KJF07] for an example from the area of process scheduling.

Discussion. Semantic Web technology can be seen as a promising base technology for future, more intelligent error-handling techniques in web service processes which are based on automated service discovery or dynamic service composition.

However, several challenges remain in practical environments. First, although OWL today is the the de-facto standard for ontology representation, there exists no agreed-upon standard for describing web service semantics. The different proposals (OWL-S, WSMO, WSDL-S) have been submitted for standardization already in 2004 and 2005, but no "winner" has been determined yet.

Beside this problem of standardizing the language, most applications domains suffer from the problem that no shared domain ontology exists or that several competing pseudo-standards exist. Consider again the example from the travel planning domain. While there exist several classification schemes such

as UNSPSC, the sector is far from having a common ontology that unambiguously describes what the effects of a "flight reservation" or a "hotel booking" are. Thus, semantics-based error recovery is only possible if (a) the involved companies tightly co-operate and agree on a detailed shared ontology or (b) domain-specific standards such as the MPEG-7 specification in the context of the work in [JL07] already exist.

Next, as mentioned in [Sri03], there are some fundamental differences between BPEL-based process models and the representation mechanism used in AI planning. BPEL processes describe a global picture of the flow of activities, which may contain complex control structures and so forth. With AI-planning techniques, one needs to model individual activities and think of appropriate preconditions and effects that are required to express rather simple precedence constraints.

Another open issue is the question of how to model side-constraints that have to apply to the overall process, which cannot be easily captured by the IOPE-style action descriptions. There might be for example several online airline booking services available, which are basically exchangeable with respect to their functionality. Due to internal business rules, it might however only be allowed to use a certain subset of those booking services in certain situations.

Finally, also questions of scalability of the planning and re-planning approaches to web service composition and the computation of repair plans have not been answered so far. Some more practical problems of web service composition (which are not even semantically-enriched ones) are also discussed in the "reality-check" paper of Lu et al. in [LYRS07].

5 Summary

In contrast to classical workflow processes, today's web service processes require additional attention with respect to exception handling and recovery because of their inherent distributed and open nature. Problematic cases for instance include situations where a remote service is simply unreachable or where detailed information about a remote exception is simply unavailable.

In this chapter, we have discussed various mechanisms to handle exceptions in web service processes. Today's de-facto process modeling "standard" technology (BPEL and BPMN) is influenced by research in the area of earlier workflow systems and relies on the "catch-and-handle" exception handling pattern known from programming languages as well as on the concepts of transactions and compensating actions. In order to support modular and compact process models that do not mix the regular execution flow with error-handling procedures, some languages such as YAWL provide special model structuring mechanisms and the opportunity to dynamically change the execution flow at run-time.

In the final sections, an outline of more intelligent error-handling mechanism of current web service process technology was given. Most recent approaches aim to avoid explicit error-handling models and exploit additional knowledge to calculate problem recovery actions automatically. These recent proposals include

for example more advanced debugging aids based on Model-based Diagnosis as well as techniques for the computation of corresponding repair plans. In addition, the opportunities for automated web service discovery and re-planning as a special form of service composition that come along with the emergence of Semantic Web technology have been discussed in this chapter.

References

[ACG+05] L. Ardissono, L. Console, A. Goy, G. Petrone, C. Picardi, M. Segnan, and D. T. Dupre, *Enhancing web services with diagnostic capabilities*, Proc. 3rd Europ. Conf. on Web Services (Växjö, Sweden), 2005, p. 182.

[AFG+06] L. Ardissono, R. Furnari, A. Goy, G. Petrone, and M. Segnan, *Fault tolerant web service orchestration by means of diagnosis*, EWSA 2006, LNCS, vol. 4344, Springer, 2006, pp. 2–16.

[AtHEvdA06] M. Adams, A. H. M. ter Hofstede, D. Edmond, and W. M. P. van der Aalst, *Worklets: A service-oriented implementation of dynamic flexibility in workflows*, Proc. CooPIS 2006, 2006, pp. 291–308.

[AtHvdAE07] M. Adams, A.H.M. ter Hofstede, W.M.P. van der Aalst, and D. Edmond, *Dynamic, extensible and context-aware exception handling for workflow*, Proc. CoopIS 2007 (Algarve, Portugal), 2007.

[BLHL01] T. Berners-Lee, J. Hendler, and O. Lassila, *The semantic web*, Scientific American (2001), 34–43.

[CCPP99] F. Casati, S. Ceri, S. Paraboschi, and G. Pozzi, *Specification and implementation of exceptions in workflow management systems*, ACM Transactions on Database Systems **24** (1999), no. 3, 405–451.

[CDP08] C. Combi, F. Daniel, and G. Pozzi, *XPDL Enabled Cross-Product Exception Handling for WfMSs*, 2008 BPM and Workflow Handbook (Florida) (Layna Fischer, ed.), Future Strategies, Inc, 2008, pp. 177–186.

[CJ90] P. Compton and R. Jansen, *Knowledge in context: a strategy for expert system maintenance*, Proc. 2nd Australian Joint Conference on Artificial Intelligence, 1990, pp. 292–306.

[DKLW07] G. Decker, O. Kopp, F. Leymann, and M. Weske, *Bpel4chor: Extending bpel for modeling choreographies*, ICWS'2007 (Salt Lake City, Utah), 2007, pp. 296–303.

[EFL+04] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres, *A logic programming approach to knowledge-state planning: Semantics and complexity*, ACM Transactions on Computational Logic **5** (2004), no. 2, 206–263.

[EL96] J. Eder and W. Liebhart, *Workflow recovery*, Proc. 1st IFCIS International Conference on Cooperative Information Systems, 1996, p. 124.

[EL97] ———, *Workflow transactions*, Workflow Handbook, John Wiley, 1997, pp. 157–163.

[ES07] J. Euzenat and P. Shvaiko, *Ontology matching*, Springer-Verlag, Heidelberg, 2007.

[FFJS04] A. Felfernig, G. Friedrich, D. Jannach, and M. Stumptner, *Consistency-based diagnosis of configuration knowledge bases*, Artificial Intelligence **152** (2004), no. 2, 213–234.

[FFM+10] G. Friedrich, M. Fugini, E. Mussi, B. Pernici, and G. Tagni, *Exception handling for repair in service-based processes*, IEEE Trans. Software Eng. **36** (2010), no. 2, 198–215.

[FI08a] G. Friedrich and V. Ivanchenko, *Diagnosis from first principles for workflow executions*, Technical Report 2008/002. Institute of Applied Informatics, University Klagenfurt, Austria, 2008.

[FI08b] ———, *Model-based repair of web service processes*, Technical Report 2008/001. Institute of Applied Informatics, University Klagenfurt, Austria, 2008.

[FMS10] G. Friedrich, W. Mayer, and M. Stumptner, *Diagnosing process trajectories under partially known behavior*, Proc. ECAI 2010 (Lisbon, Portugal), 2010, pp. 111–116.

[FN95] R. E. Fikes and N. J. Nilsson, *Strips: a new approach to the application of theorem proving to problem solving*, In: Computation & Intelligence: collected readings, AAAI Press, 1995, pp. 429–446.

[FSW99] G. Friedrich, M. Stumptner, and F. Wotawa, *Model-based diagnosis of hardware designs*, Artificial Intelligence **111** (1999), no. 1-2, 3–39.

[Gla08] O. Glassey, *A case study on process modelling - three questions and three techniques*, Decision Support Systems **44** (2008), no. 4, 842–853.

[GM06] N. Guelfi and A. Mammar, *A formal framework to generate XPDL specifications from uml activity diagrams*, Proc. SAC'06, 2006, pp. 1224–1231.

[HA00] C. Hagen and G. Alonso, *Exception handling in workflow management systems*, IEEE Trans. Softw. Eng. **26** (2000), no. 10, 943–958.

[HBP07] J. Hoffmann, P. Bertoli, and M. Pistore, *Web service composition as planning, revisited: In between background theories and initial state uncertainty*, Proc. AAAI'07 (Vancouver, BC), 2007, pp. 1013–1018.

[HH06] O. Halvorsen and O. Haugen, *Proposed notation for exception handling in UML 2 sequence diagrams*, Proc. 18th Australian Software Engineering Conference (Sydney, Australia), 2006, pp. 29–40.

[Hof05] A. H. M. Ter Hofstede, *Yawl: yet another workflow language*, Information Systems **30** (2005), 245–275.

[JL07] D. Jannach and K. Leopold, *Knowledge-based multimedia adaptation for ubiquitous multimedia consumption*, Journal of Network and Computer Applications **30** (2007), no. 3, 958–982.

[JLTH06] D. Jannach, K. Leopold, C. Timmerer, and H. Hellwagner, *A knowledge-based framework for multimedia adaptation*, Applied Intelligence **24** (2006), no. 2, 109–125.

[KJF07] J. Kuster, D. Jannach, and G. Friedrich, *Handling alternative activities in resource-constrained project scheduling problems*, Proceedings IJCAI'07 (Hyderabad, India), 2007, pp. 1960–1965.

[KSF07] J. Kopecký, E. Paslaru Bontas Simperl, and D. Fensel, *Semantic web service offer discovery*, Proc. Workshop on Service Matchmaking and Resource Retrieval in the Semantic Web (SMRR 2007) co-located with ISWC 2007 + ASWC 2007 (Busan, South Korea), 2007.

[LPF+06] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello, *The dlv system for knowledge representation and reasoning*, ACM Transactions on Computational Logic **7** (2006), no. 3, 499–562.

[LYRS07] J. Lu, Y. Yu, D. Roy, and D. Saha, *Web service composition: A reality check*, Proc. WISE 2007 (Nancy, France), 2007, pp. 523–532.

[MMP06] S. Modafferi, E. Mussi, and B. Pernici, *SH-BPEL: a self-healing plug-in for WS-BPEL engines*, Proc. Workshop on Middleware for Service Oriented Computing, 2006, pp. 48–53.

[MSZ01] S. A. McIlraith, T. C. Son, and H. Zeng, *Semantic web services*, IEEE Intelligent Systems **16** (2001), no. 2, 46–53.

[PC96] L. Pryor and G. Collins, *Planning for contingencies: A decision-based approach*, Journal of Artificial Intelligence Research **4** (1996), 287–339.

[Pee05] Joachim Peer, *A pop-based replanning agent for automatic web service composition*, Proc. ESWC 2005 (Heraklion, Crete), 2005, pp. 47–61.

[RD98] M. Reichert and P. Dadam, *Adept$_{flex}$ - supporting dynamic changes of workflows without losing control*, Journal of Intelligent Information Systems **10** (1998), no. 2, 93–129.

[Rei87] R. Reiter, *A theory of diagnosis from first principles*, Artificial Intelligence **32** (1987), no. 1, 57–95.

[RvdAtH06] N. Russell, W. van der Aalst, and A. ter Hofstede, *Workflow exception patterns*, Proc. CAiSE 2006 (Luxembourg), 2006.

[SPW+04] E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau, *HTN planning for web service composition using SHOP2*, Proc. ISCW 2004 (Sanibel Island, Florida), October 2004, pp. 377–396.

[SR93] A. Sheth and M. Rusinkiewicz, *On transactional workflows*, IEEE Data Engineering Bulletin **16** (1993), 3–34.

[Sri03] B. Srivastava, *Web service composition - current solutions and open problems*, Prof. ICAPS 2003 Workshop on Planning for Web Services (Trento, Italy), 2003, pp. 28–35.

[STA05] A. W. Scheer, O. Thomas, and O. Adam, *Process modeling using event-driven process chains*, Process-Aware Information Systems (M. Dumas, W. M. P. van der Aalst, and A. H. M. ter Hofstede, eds.), Wiley, 2005, pp. 119–145.

[TAH07] V. Tsetsos, C. Anagnostopoulos, and S. Hadjiefthymiades, *Semantic web service discovery: Methods, algorithms and tools*, Semantic Web Services: Theory, Tools and Applications (J. Cardoso, ed.), IDEA, 2007, pp. 240–280.

[vNEV07] D. van Nieuwenborgh, T. Eiter, and D. Vermeir, *Conditional planning with external functions*, Proc. LPNMR'07 (Tempe, AZ, USA), 2007, pp. 214–227.

[Wei81] M. Weiser, *Program slicing*, Proc. 5th International Conference on Software Engineering (Piscataway, NJ, USA), 1981, pp. 439–449.

[Wot02] F. Wotawa, *On the relationship between model-based debugging and program slicing*, Artificial Intelligence **135** (2002), no. 1-2, 125–143.

[YD07] Y. Yan and P. Dague, *Modeling and diagnosing orchestrated web service processes*, Proc. ICWS 2007 (Salt Lake City, Utah), 2007, pp. 51–59.