Dynamic and Extensible Exception Handling for Workflows: A Service-Oriented Implementation

Michael Adams[1], Arthur H. M. ter Hofstede[1], David Edmond[1], and Wil M. P. van der Aalst[1,2]

[1] Business Process Management Group
Queensland University of Technology, Brisbane, Australia
{m3.adams,a.terhofstede,d.edmond}@qut.edu.au
[2] Department of Mathematics and Computer Science
Eindhoven University of Technology, Eindhoven, The Netherlands
w.m.p.v.d.aalst@tue.nl

Abstract. This paper presents the realisation, using a Service Oriented Architecture, of an approach for dynamic, flexible and extensible exception handling in workflows, based not on proprietary frameworks, but on accepted ideas of how people actually work. The approach utilises an established framework for workflow flexibility called *worklets* and a detailed taxonomy of workflow exception patterns to provide an extensible repertoire of self-contained exception-handling processes which may be applied at the task, case or specification levels, and from which a dynamic runtime selection is made depending on the context of the exception and the particular work instance. Both expected and unexpected exceptions are catered for in real time, so that 'manual handling' of the exception is avoided.

Key words : workflow exception handling, workflow flexibility, service oriented architecture, worklet

1 Introduction

Workflow management systems (WfMS) are used to configure and control structured business processes from which well-defined workflow models and instances can be derived [1, 2]. However, the proprietary process definition frameworks imposed by WfMSs make it difficult to support (i) dynamic evolution (i.e. modifying process definitions during execution) following unexpected or developmental change in the business processes being modelled [3]; and (ii) exceptions, or deviations from the prescribed process model at runtime [4–6].

For exceptions, the accepted practice is that if an exception can conceivably be anticipated, then it should be included in the process model. However, this approach can lead to very complex models, much of which will never be executed in most cases, and adds orders-of-magnitude complexities to workflow logic; mixing business logic with exception handling routines complicates the verification and modification of both [7], in addition to rendering the model almost unintelligible to most stakeholders.

Conversely, if an exception occurs that is unexpected, the model is deemed to be simply deficient, and thus needs to be amended to include the previously unimagined event (see for example [8]). This approach, however, tends to gloss over the frequency of such events and the costs involved with their correction. Most often, suspension of execution while the deviation is handled manually or the termination of the entire process instance are the only available options, but since most processes are long and complex, neither option presents a satisfactory solution [9]. Manual handling incurs an added penalty: the corrective actions undertaken are not added to 'organisational memory' [10, 11], and so natural process evolution is not incorporated into future iterations of the process. Associated problems include those of migration, synchronisation and version control [4].

These limitations imply that a large subset of business processes do not easily map to the rigid modelling structures provided [12], due to the lack of flexibility inherent in a framework that, by definition, imposes rigidity. This is further supported by our work on process mining. When considering processes where people are expected to execute tasks in a structured way but are not forced by some workflow system, process mining shows that the processes are much more dynamic than expected, that is, people tend to deviate from the "normal flow", often with good reasons.

Thus, process models are 'system-centric', or *straight-jacketed* [1] into the supplied framework, rather than truly reflecting the way work is actually performed [13]. As a result, users are forced to work outside of the system, and/or constantly revise the static process model, in order to successfully support their activities, thereby negating the efficiency gains sought by implementing a workflow solution in the first place.

Since the mid-nineties many researchers have worked on problems related to workflow flexibility and exception handling (cf. Section 7). This paper is based on and extends the *'worklets'* approach described in [14] and [15] and applies the classification of workflow exception patterns from [16]. It introduces a realisation of a service that utilises an extensible repertoire of self-contained exception handling processes and associated selection rules, grounded in a formal set of work practice principles called *Activity Theory*, to support the flexible modelling, analysis, enactment and support of business processes. This approach directly provides for dynamic change and process evolution without having to resort to off-system intervention and/or system downtime. It has been implemented as a discrete service for the well-known, open-source workflow environment YAWL [17, 18] using a Service Oriented Architecture (SOA), and as such its applicability is in no way limited to that environment. Also, being open-source, it is freely available for use and extension.

The paper illustrates aspects of the approach throughout using the organisation of a rock concert as an example process and is organised as follows: Section 2 provides a brief overview of the theoretical underpinnings of the approach. Section 3 provides an overview of the design and operation and service, while Section 4 details the service architecture. Section 5 discusses exception types

handled by the service and the definition of exception handling processes. Section 6 describes how the approach utilises *Ripple Down Rules* (RDR) to achieve contextual, dynamic selection of handling processes at runtime. Section 7 discusses related work, and finally Section 8 outlines future directions and concludes the paper.

2 Theoretical Framework

In [19], we undertook a detailed study of *Activity Theory*, a broad collective of theorising and research in organised human activity (cf. [20–22]) and derived from it a set of principles that describe the nature of participation in organisational work practices. Briefly, the principles relevant to this paper are:

1. *Activities* (i.e. work processes) are *hierarchical* (consist of one or more actions), *communal* (involve a community of participants working towards a common objective), *contextual* (conditions and circumstances deeply affect the way the objective is achieved), *dynamic* (evolve asynchronously), and *mediated* (by tools, rules and divisions of labour).
2. *Actions* (i.e. tasks) are undertaken and understood contextually. A *repertoire* of applicable actions is maintained and made available to each activity, which is performed by making contextual choices from the repertoire.
3. A *work plan* is not a prescription of work to be performed, but merely a guide which may be modified during execution depending on context.
4. *Deviations* from a plan will naturally occur with every execution, giving rise to learning experiences which can then be incorporated into future instantiations of the plan.

Consideration of these principles has delivered a discrete service that transforms otherwise static workflow processes into fully flexible and dynamically extensible process instances by offering full support for realtime handling of both expected and unexpected exceptions, using a Service-Oriented Architecture. The service:

- regards the process model as a guide to an activity's objective, rather than a prescription for it;
- provides a repertoire (or catalogue) of applicable actions to be made available at each execution of a process model;
- provides for choices to be made dynamically from the repertoire at runtime by considering the specific context of the executing instance; and
- allows the repertoire of actions to be dynamically extended at runtime, thus incorporating unexpected process deviations, not only for the current instance, but for other current and future instantiations of the process model, leading to natural process evolution.

As detailed in the following sections, the service has been implemented directly on top of an Activity Theory framework, and thus provides workflow support for processes from a wide variety of work environments.

3 Worklet Service Description

The *Worklet Service* (essentially, a *worklet* is a small, discrete workflow process that acts as a late-bound sub-net for an enabled workitem) comprises two distinct but complementary sub-services: a *Selection sub-Service*, which enables dynamic flexibility for YAWL process instances [14]; and an *Exception sub-Service* (the focus of this paper), which provides facilities to handle both expected and unexpected process exceptions (i.e. events and occurrences that may happen during the life of a process instance that are not explicitly modelled within the process) at runtime.

The Selection Service: The Selection Service enables flexibility by allowing a process designer to designate certain workitems to each be substituted at runtime with a dynamically selected *worklet*, which contextually handles one specific task in a larger, composite process activity. Each worklet instance is dynamically selected and invoked at runtime and may be designed and provided to the Selection Service at any time, as opposed to a static sub-process that must be defined at the same time as, and remains a static part of, the main process model.

An extensible repertoire of worklets is maintained by the Service for each task in a specification. Each time the Service is invoked for a workitem, a choice is made from the repertoire based on the contextual data values within the workitem, using an extensible set of rules to determine the most appropriate substitution.

The workitem is checked out of the workflow enactment engine, the corresponding data inputs of the original workitem are mapped to the inputs of the worklet, and the selected worklet is launched as a separate case. When the worklet has completed, its output data is mapped back to the original workitem, which is then checked back into the engine, allowing the original process to continue.

An extensive discussion of the implementation of the Worklet Selection Service may be found in [14].

The Exception Service: Virtually every process instance (even if it follows a highly structured process definition) will experience some kind of exception (or deviation) during its execution. It may be that these events are known to occur in a small number of cases, but not often enough to warrant their inclusion in the process model; or they may be things that were never expected to occur (or may be never even imagined could occur). In any case, when they do happen, since they are not included in the process model, they must be handled 'offline' before processing can continue (and the way they are handled is rarely recorded). In some cases, the process model will be later modified to capture this unforeseen event, which involves an, often large, organisational cost (downtime, remodelling, testing and so on), or in certain circumstances the entire process must be aborted.

Alternately, an attempt might be made to include every possible twist and turn into the process model so that when such events occur, there is a branch in

the process to take care of it. This approach often leads to very complex models where much of the original business logic is obscured, and doesn't avoid the same problems when the next unexpected exception occurs.

The Exception Service addresses these problems by allowing designers to define exception handling processes (called *exlets*) for parent workflow instances to be invoked when certain events occur and thereby allow the process to continue unhindered. Additionally, exlets for unexpected exceptions may be added at runtime, and such handling methods automatically become an implicit part of the process specification for all current and future instances of the process, which provides for continuous evolution of the process while avoiding the need to modify the original process definition.

The Exception Service uses the same repertoire and dynamic rules approach as the Selection Service. There are, however, two fundamental differences between the two sub-services. First, where the Selection Service selects a worklet as the result of satisfying a rule in a rule set, the result of an Exception Service selection is an exlet (which may contain a worklet to be executed as a compensation process – see Section 5). Second, while the Selection Service is invoked for certain nominated tasks in a YAWL process, the Exception Service, when enabled, is invoked for *every* case and task executed by the YAWL engine, and will detect and handle up to ten different kinds of process exceptions (these exception types are described in Section 5). As part of the exlet, a process designer may choose from various actions (such as cancelling, suspending, completing, failing and restarting) and apply them at a workitem, case and/or specification level. And, since the exlets can include compensatory worklets, the original parent process model only needs to reveal the actual business logic for the process, while the repertoire of exlets grows as new exceptions arise or different ways of handling exceptions are formulated. Table 1 summarises the differences between the two sub-services (the interfaces are described in the next section).

Table 1. Summary of Service Actions

Cause	Interface	Selection	Action Returned
Workitem Enabled	B	Case & item context data	Worklet
Internal Exception	X	Exception type and Case & item context data	Exlet
External Exception	–	Exception type and Case & item context data	Exlet

An extensible repertoire of exlets is maintained by the service for each workflow specification, and may be applied at the workitem, case or specification level. Each time the service is notified of an event or checkpoint, the service first determines whether an exception has in fact occurred, and if so makes a choice from the repertoire based on the type of exception and the data attributes and values associated with the workitem/case, using a set of rules to select the most appropriate exlet to execute (see Section 6).

If the exlet contains a compensation action (i.e. a worklet to executed as a compensatory process) it is run as a separate case in the enactment engine, so that from an engine perspective, the worklet and its 'parent' (i.e. the process that invoked the exception) are two distinct, unrelated cases. The service tracks the relationships, data mappings and synchronisations between cases, and maintains a process log that may be combined with the engine's process logs via case identifiers to provide a complete operational history of each process. Figure 1 shows the relationship between a 'parent' process, an exlet repertoire and a compensatory worklet, using the *Organise Concert* example.

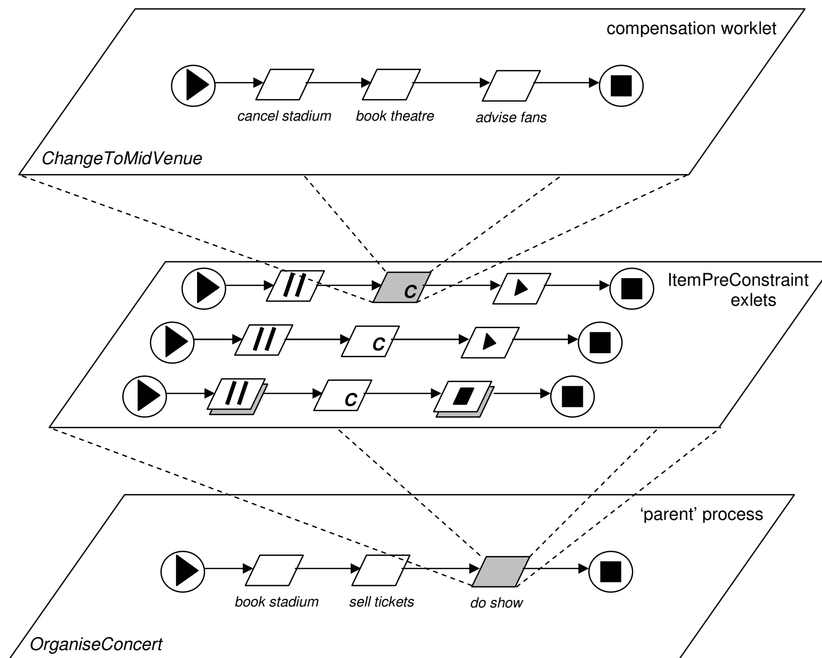

Fig. 1. Process – Exlet – Worklet Hierarchy

Any number of exlets can form the repertoire of an individual task or case. An exlet may be a member of one or more repertoires – that is, it may be re-used for several distinct tasks or cases within and across process specifications.

The repertoire for a task or case can be added to at any time, as can the rules base used, *including while the parent process is executing.* Thus the service provides for dynamic ad-hoc change, exception handling and process evolution, without having to resort to off-system intervention and/or system downtime, and avoiding the need to modify the original process specification.

The Selection and Exception sub-services can be used in combination within particular case instances to achieve dynamic flexibility *and* exception handling simultaneously. The Worklet Service is extremely adaptable and multi-faceted, and allows a designer to provide tailor-made solutions to runtime process exceptions.

4 Service Architecture

The Worklet Service has been implemented as a YAWL Custom Service [17, 18]. The YAWL environment was chosen as the implementation platform since it provides a very powerful and expressive workflow language based on the workflow patterns identified in [23], together with a formal semantics. It also provides a workflow enactment engine, and an editor for process model creation, that support the control flow, data and (basic) resource perspectives. The YAWL environment is open-source and offers a service-oriented architecture, allowing the service to be implemented completely independent to the core engine. Thus the deployment of the Worklet Service is in no way limited to the YAWL environment, but may be ported to other environments (for example, BPEL engines) by making the necessary links in the service interface. As such, this implementation may also be seen as a case study in service-oriented computing whereby dynamic exception handling for workflows, orthogonal to the underlying workflow language, is provided.

Figure 2 shows the external architecture of the Worklet Service. The YAWL system allows workflow instances and external services to interact with each other in order to delegate work, to signal the creation and completion of process instances and workitems, or to notify of certain events or changes in the status of existing workitems and cases. These services interact with the YAWL engine across a number of interfaces designed for particular purposes, supporting the ability to send and receive both messages and XML data to and from the engine. Three interfaces are used by the Worklet Service (see Figure 2):

- Interface A provides endpoints for process definition, administration and monitoring [18] – the service uses Interface A to upload worklet specifications to the engine;
- Interface B provides endpoints for client and invoked applications and workflow interoperability [18] – used by the service for connecting to the engine, to start and cancel case instances, and to check workitems in and out of the engine after interrogating their associated data; and
- Interface X ('X' for 'e**X**ception'), which has been designed to allow the engine to notify custom services of certain events and checkpoints during the

execution of each process instance where process exceptions either may have occurred or should be tested for. Thus Interface X provides the service with the necessary mechanisms to dynamically capture and handle process exceptions.

In fact, Interface X was created to enable the Exception sub-service to be built. However, one of the overriding design objectives was that the interface should be structured for generic application – that is, it can be applied by a variety of services that wish to make use of checkpoint and/or event notifications during process executions.

Since it only makes sense to have one custom service acting as an exception handling service at any one time, services that implement Interface X have two distinct states – *enabled* and *disabled*. When enabled, the engine generates notifications for *every* process instance it executes – that is, the engine makes no decisions about whether a particular process should generate the notifications or not. Thus it is the responsibility of the designer of the custom service to determine how best to deal with the notifications. When the service is disabled, the engine generates no notifications across the interface. Enabling and disabling an Interface X custom service is achieved via parameter setting in a configuration file.

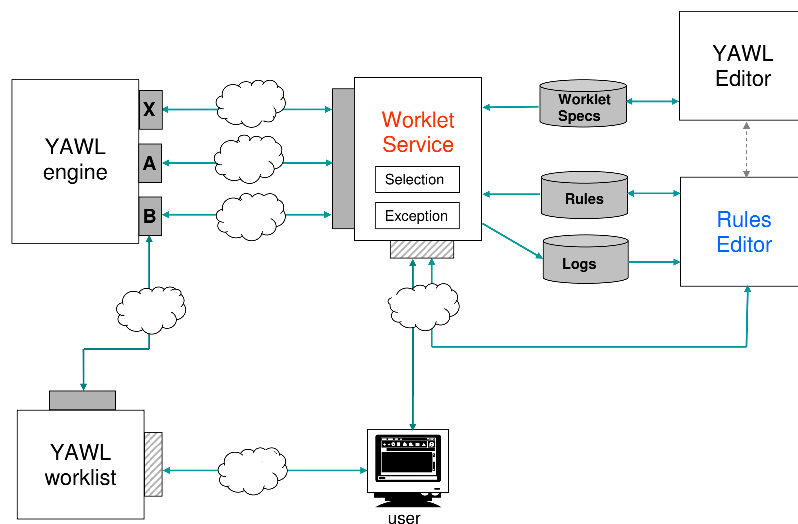

Fig. 2. External Architecture of the Worklet Service

The entities 'Worklet specs', 'Rules' and 'Logs' in Figure 2 comprise the *worklet repository*. The service uses the repository to store rule sets, worklet specifications for uploading to the engine, and generated process and audit logs. The YAWL editor is used to create new worklet specifications, and may be invoked from the Rules Editor, which is used to create new or augment existing

rule sets, making use of certain selection logs to do so, and may communicate with the Worklet Service via a JSP/Servlet interface to override worklet selections following rule set additions (see Section 6). The service also provides servlet pages that allow users to directly communicate with the service to raise external exceptions and carry out administration tasks.

5 Exception Types and Handling Primitives

This section introduces the ten different types of process exception that have been identified, seven of which are supported by the current version of the Worklet Service. It then describes the handling primitives that may be used to form an exception handling process (i.e. an exlet) for the notified exception event. The exception types and primitives described here are based on and extend from those identified by Russell et. al., who define a rigorous classification framework for workflow exception handling independent of specific modelling approaches or technologies [16].

The Exception sub-service maintains a set of rules (described in detail in Section 6) that is used to determine which exlet, if any, to invoke. If there are no rules defined for a certain exception type for a specification, the exception event is simply ignored by the service. Thus rules are needed only for those exception events that are desired to be handled for a particular task and/or specification.

5.1 Exception Types

Constraint Types Constraints are rules that are applied to a workitem or case immediately before and after execution of that workitem or case. Thus, there are four types of constraint exception:

- *CasePreConstraint* - case-level pre-constraint rules are checked when each case instance begins execution;
- *ItemPreConstraint* - item-level pre-constraint rules are checked when each workitem in a case becomes enabled (i.e. ready to be checked out);
- *ItemPostConstraint* - item-level post-constraint rules are checked when each workitem moves to a completed status; and
- *CasePostConstraint* - case-level post constraint rules are checked when a case completes.

The service receives notification from the YAWL Engine when each of these constraint events occur within each case, then checks the rule set associated with the case to determine, firstly, if there are any rules of that exception type defined for the case, and if so, if any of the rules evaluate to true using the contextual data of the case or workitem. If the rule set finds a rule that evaluates to true for the exception type and data, an associated exlet is selected and invoked.

TimeOut A timeout event occurs when a workitem is linked to the YAWL Time Service and the deadline set for that workitem is reached. In this case, the YAWL Engine notifies the Worklet Service of the timeout event, and passes to the service a reference to the workitem and each of the other workitems that were running in parallel with it. Therefore, timeout rules may be defined for each of the workitems affected by the timeout (including the actual timed out workitem itself).

Externally Triggered Types Externally triggered exceptions occur, not through the case's data parameters, but because of an occurrence outside of the process instance that has an effect on the continuing execution of the process. Thus, these events are triggered by a user; depending on the actual event and the context of the case or workitem, a particular exlet will be invoked. There are two types of external exceptions, CaseExternalTrigger (for case-level events) and ItemExternalTrigger (for item-level events).

These seven types of exceptions are supported by our current implementation. Three more exception types have been identified but are not yet supported:

ItemAbort An ItemAbort event occurs when a workitem being handled by an external program (as opposed to a human user) reports that the program has aborted before completion.

ResourceUnavailable This event occurs when an attempt has been made to allocate a workitem to a resource and the resource reports that it is unable to accept the allocation or the allocation cannot proceed.

ConstraintViolation This event occurs when a data constraint has been violated for a workitem *during* its execution (as opposed to pre- or post- execution).

5.2 Exception Handling Primitives

When any of the above exception events occur, an appropriate exlet, if defined, will be invoked. Each exlet may contain any number of steps, or *primitives*, and is defined graphically using the Worklet Rules Editor.

An example of a definition of an exlet in the Rules Editor can be seen in Figure 3. On the left of the graphical editor is the set of primitives that may be used. The available primitives (reading left-to-right, top-to-bottom) are:

- *Remove WorkItem*: removes (or cancels) the workitem; execution ends, and the workitem is marked with a status of cancelled. No further execution occurs on the process path that contains the workitem.
- *Remove Case*: removes the case. Case execution ends.
- *Remove All Cases*: removes all case instances for the specification in which the workitem is defined, or of which the case is an instance.

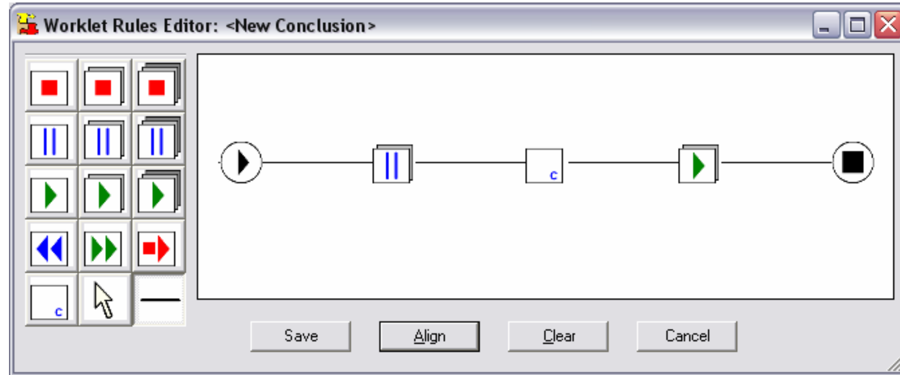

Fig. 3. Example Handler Process in the Rules Editor

- *Suspend WorkItem*: suspends (or pauses) execution of a workitem, until it is continued, restarted, cancelled, failed or completed, or the case that contains the workitem is cancelled or completed.
- *Suspend Case*: suspends all 'live' workitems in the current case instance (a live workitem has a status of fired, enabled or executing), effectively suspending execution of the entire case.
- *Suspend All Cases*: suspends all 'live' workitems in all of the currently executing instances of the specification in which the workitem is defined, effectively suspending all running cases of the specification.
- *Continue WorkItem*: un-suspends (or continues) execution of the previously suspended workitem.
- *Continue Case*: un-suspends execution of all previously suspended workitems for the case, effectively continuing case execution.
- *Continue All Cases*: un-suspends execution of all workitems previously suspended for all cases of the specification in which the workitem is defined or of which the case is an instance, effectively continuing all previously suspended cases of the specification.
- *Restart WorkItem*: rewinds workitem execution back to its start. Resets the workitem's data values to those it had when it began execution.
- *Force Complete WorkItem*: completes a 'live' workitem. Execution of the workitem ends, and the workitem is marked with a status of *ForcedComplete*, which is regarded as a successful completion, rather than a cancellation or failure. Execution proceeds to the next workitem on the process path.
- *Force Fail WorkItem*: fails a 'live' workitem. Execution of the workitem ends, and the workitem is marked with a status of *Failed*, which is regarded as an unsuccessful completion, but not as a cancellation – execution proceeds to the next workitem on the process path.
- *Compensate*: runs a compensatory process (i.e. a worklet). Depending on previous primitives, the worklet may execute simultaneously to the parent case, or execute while the parent is suspended.

Worklets can in turn invoke child worklets to any depth. The primitives 'Suspend All Cases', 'Continue All Cases' and 'Remove All Cases' may be edited when being added to an exlet definition in the Rules Editor so that their action is restricted to ancestor cases only. Ancestor cases are those in a hierarchy of worklets back to the original parent case – that is, where a process invokes an exlet which invokes a compensatory worklet which in turn invokes an exlet, and so on. Also, the 'continue' primitives are applied only to those workitems and cases that were previously suspended by the same exlet.

A compensation primitive may contain an array of one or more worklets – when multiple worklets are defined for a compensation primitive via the Rules Editor, they are launched concurrently as a composite compensatory action when the exlet is executed. Execution moves to the next primitive in the exlet when all worklets have completed.

In the same manner as the Selection sub-service, the Exception sub-service also supports data mapping from a case to a compensatory worklet and back again. For example, if a certain variable has a value that prevents a case instance from continuing, a worklet can be run as a compensation, during which a new value can be assigned to the variable and that new value mapped back to the parent case, so that it may continue execution.

Referring back to Figure 1, the centre tier shows the exlets defined for ItemPreConstraint violations. As mentioned above, there may actually be up to eleven different members of this tier. Also, each exlet may refer to a different set of compensatory processes, or worklets, and so at any point there may be several worklets operating on the upper tier.

Rollback: A further primitive identified by Russell et. al. is 'Rollback' [16], where the execution of the process may be unwound back to a specified point and all changes to the case's data from that point forward are undone. The term 'rollback' is taken from database processing, where it serves the essential purpose of reverting the database to a previous stable state if, for some reason, a problem occurs during an update. Thus, rollback certainly applies in terms of workflow systems at the *transactional* level. However, for this implementation we considered that a rollback action serves no real purpose at the control-flow level and so has not been included. For tasks that have already completed, erasing the outcomes of those tasks as if they had never been carried out is counterproductive; better to execute a compensation exlet that corrects the problem so that both the original and corrective actions are maintained – that is, a *redo* is more appropriate than an *undo* at the control-flow level. In so doing, a complete picture of the entire process is available. There is enough flexibility inherit in the primitives above to accommodate any kind of compensatory action. For example, if a loan is approved before it becomes evident that a error of judgement has been made by the approving officer, it is better to run some compensation to redo the approval process again (so that a record of both approval processes remains), rather than rollback the approval process, and thus lose the details of the original approval.

6 Contextual Selection of Exlets

The runtime selection of an appropriate exlet relies on the type of exception that has occurred and the relevant context of each case instance, derived from case and historical data. The selection process is achieved through the use of modified *Ripple Down Rules* (RDR), which comprise a hierarchical set of rules with associated exceptions, first devised by Compton and Jansen [24]. The fundamental feature of RDR is that it avoids the difficulties inherent in attempting to compile, *a-priori*, a systematic understanding, organisation and assembly of all knowledge in a particular domain. Instead, it allows for general rules to be defined first with refinements added later as the need arises [25].

Any specification may have an associated rule set, which consists of a collection of RDR trees stored as XML data. Each RDR tree is a collection of simple rules of the form "if *condition* then *conclusion*", conceptually arranged in a binary tree structure (see Fig. 4). When a rule tree is queried, it is traversed from the root node of the tree along the branches, each node having its condition evaluated along the way. If a node's condition evaluates to *True*, and it has a true child (that is, it has a child node connected on a *True* branch), then that child node's condition is also evaluated. If a node's condition evaluates to *False*, and there is a false child, then that child node's condition is evaluated [26]. When a terminal node is reached, if its condition evaluates to *True* then that conclusion is returned as the result of the tree traversal; if it evaluates to *False*, then the last node in the traversal that evaluated to *True* is returned as the result.

Effectively, each rule node on the true branch of its parent node is an exception rule of the more general one of its parent (that is, it is a *refinement* of the parent rule), while each rule node on the false branch of its parent node is an "else" rule to its parent (or an *alternate* to the parent rule). This tree traversal provides implied *locality* - a rule on an exception branch is tested for applicability only if its parent (next-general) rule is also applicable.

The hierarchy of a worklet rule set is (from the bottom up):

- **Rule Node**: contains the details (condition, conclusion, id, parent and so on) of one discrete ripple-down rule.
- **Rule Tree**: consists of a number of rule nodes conceptually linked in a binary tree structure.
- **Tree Set**: a set of one or more rule trees. Each tree set is specific to a particular rule type (Timeout, ExternalTrigger, etc.). The tree set of a case-level exception rule type will contain exactly one tree. The tree set of an item-level rule type will contain one rule tree for each task of the specification that has rules defined for it (not all tasks in the specification need to have a rule tree defined).
- **Rule Set**: a set of one or more tree sets representing the entire set of rules defined for a specification. Each rule set is specific to a particular specification. A rule set will contain one tree set for each rule type for which rules have been defined.

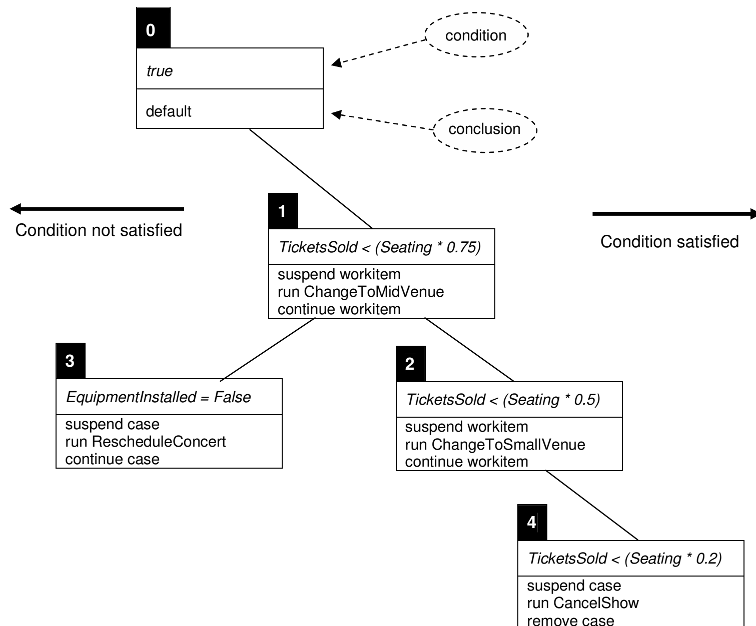

Fig. 4. Example rule tree (for OrganiseConcert ItemPreConstraint)

Each specification has a unique rule set (if any), which contains between one and eleven tree sets (or sets of rule trees), one for selection rules (used by the Selection sub-service) and one for each of the ten exception types. Three of those ten relate to case-level exceptions (i.e. CasePreConstraint, CasePostConstraint and CaseExternalTrigger) and so each of these will have at most one rule tree in the tree set. The other eight tree sets relate to workitem-level events (seven exception types plus selection), and so may have one rule tree for each task in the specification - that is, the tree sets for these eight rule types may consist of a number of rule trees.

It is not necessary to define rules for all eleven types for each specification, only for those types that are required to be handled; the occurrence any exception types that aren't defined in the rule set file are simply ignored. So, for example, if an analyst is interested only in capturing pre- and post- constraints at the workitem level, then only the ItemPreConstraint and ItemPostConstraint tree sets need to be defined (i.e. rules defined within those tree sets). Of course, rules for other event types can be added later if required.

Figure 4) shows the ItemPreConstraint rule tree for the Organise Concert example, which represents the rule tree for the exlets shown on the centre tier of Figure 1. The condition part is the rule that is evaluated, and the conclusion is the exlet selected by that rule if the condition evaluates to true.

The third task in the OrganiseConcert specification, *Do Show*, has a pre-item constraint rule tree (refer Fig. 1), and so when a workitem of the task becomes

enabled (and thus the engine notifies the service), the rule tree is queried. The Rules Editor provides a textual representation of the relevant rule tree (called the effective composite rule) as can be seen in Figure 5.

```
Effective Composite Rule
if TicketsSold < (Seating * 0.75) then suspend workitem; run worklet ChangeToMidVenue; continue workitem
  except if TicketsSold < (Seating * 0.5) then suspend workitem; run worklet ChangeToSmallVenue; continue workitem
    except if TicketsSold < (Seating * 0.2) then suspend case; run worklet CancelShow; remove case
```

Fig. 5. Effective Composite Rule for Do Show's Pre-Item Constraint Tree

When *Do Show* is enabled and the value of the case data attribute 'TicketsSold' is less than 75% of the attribute 'Seating' (i.e. the seating capacity of the venue), an exlet is run that suspends the workitem, runs the compensatory worklet ChangeToMidVenue, and then, once the worklet has completed, continues (or unsuspend) the workitem. That is, this pre-constraint exception allows organisers to change the venue of the concert to a mid-sized stadium when there are insufficient tickets sold to fill the original venue. Following the structure of the ripple-down rule, if the tickets sold are also less than 50% of the capacity, then we want instead to suspend the workitem, run the ChangeToSmallVenue worklet, and then unsuspend the workitem. Finally, if less than 20% of the tickets have been sold, we want to suspend the entire case, run a worklet to perform the tasks required to cancel the show, and then remove (i.e. cancel) the case.

The effects of a scenario where 60% of tickets have been sold can be seen in the Available Work screen of the YAWL worklist handler (Figure 6). The *Do Show* workitem is marked as 'Suspended' and thus is unable to be selected for execution, while the ChangeToMidVenue worklet has been launched and its first workitem, *Cancel Stadium*, is enabled and may be executed. The ChangeToMidVenue worklet is being treated by the YAWL Engine as just another case, and so the service receives notifications from the engine for pre-case and pre-item constraint events for the worklet also – thus the worklet may also respond to its own exception notifications.

When the ChangeToMidVenue worklet has completed, the engine will notify the service of the case completion, at which time the service completes the third and final part of the exlet, that is continuing (unsuspending) the *Do Show* workitem so that the parent case can continue. Back at the Available Work screen, the *Do Show* workitem will now be shown as enabled and thus will be able to be checked out, and will contain the data values entered in the worklet's workitems mapped back to the *Do Show* workitem – that is, the changes to venue and capacity data values captured by the worklet are now found in *Do Show's* input data.

As mentioned previously, the service also allows for external events to be handled on-system by providing a means for exceptions to be raised by users external to the process itself. The service provides a set of servlet pages that

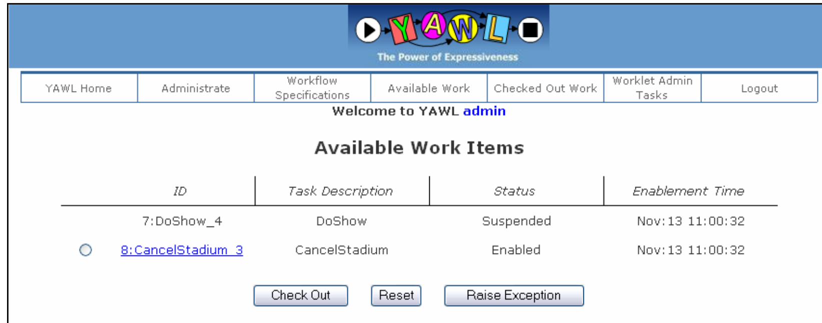

Fig. 6. Available Work Screen after ItemPreConstraint exception (Organise Concert Example)

can be invoked directly by the user via add-ins to the YAWL worklist handler, which are visible only when the service is enabled. One of the servlet pages allows a user to raise an exception directly with the service (i.e. bypassing the engine). When invoked, the Exception Service retrieves from the rule set for the selected case the list of existing external exception triggers (if any) for the case's specification. See Figure 7 for the *Raise Case-Level Exception* page listing the case-level external triggers defined for the Organise Concert specification. Note that these triggers describe events that may be considered either adverse (e.g. Band Broken Up) *or* beneficial (e.g. Ticket Sales Better Than Expected) to the current case, or may simply represent new or additional tasks that need to be carried out for the particular case instance (e.g. Band Requests Backstage Refreshments). In any case, the methods for handling these kinds of exceptions become an implicit part of the process for all future instantiations, rather than being lost.

This list contains all of the external triggers either conceived when the specification was first designed or added later as new kinds of exceptional events occurred and were added to the rule set for the specification. When a trigger is selected by the user, the conclusion for that trigger's rule is invoked by the service as an exlet for the current case.

Item-level external exceptions can be raised in a similar way. External exceptions can be raised at any time during the execution of a case - the way they are handled may depend on how far the process has progressed (via the definition of appropriate rule tree or trees which consider, as part of the rule conditionals, the status of the case and/or its workitems).

Notice that at the bottom of the list (Figure 7) the option to add a New External Exception is provided. If an unexpected external exception arises that none of the available triggers represent, a user can use that option to notify an administrator of the new exception, its context and possible ways to handle it. The administrator can then create a new exlet in the Rules Editor and, from the Editor, connect directly to the service to launch the new exlet for the parent case.

Fig. 7. Raise Case-Level Exception Screen (Organise Concert example)

New exlets for unexpected internal exceptions are raised and launched using the same approach as that described for the Selection sub-service, as detailed in [14].

7 Related Work

Since the mid-nineties much research has been carried out on issues related to exception handling in workflow management systems. While it is not the intention of this paper to provide a complete overview of the work done in this area, reference is made here to a number of quite different approaches; see [16] for a more systematic overview, where different tools are evaluated with respect to their exception handing capabilities using a patterns-based approach.

Generally, commercial workflow management systems provide only basic support for handling exceptions [9, 27] (besides modelling them directly in the main 'business logic'), and each deals with them in a proprietary manner; they typically require the model to be fully defined before it can be instantiated, and changes must be incorporated by modifying the model statically. Staffware provides constructs called *event nodes*, from which a separate pre-defined exception handling path or sequence can be activated when an exception occurs. It may also suspend a process either indefinitely or wait until a timeout occurs. If a work item cannot be processed it is forwarded to a 'default exception queue' where it may be manually purged or re-submitted. COSA provides for the definition of external 'triggers' or events that may be used to start a sub-process. All events and sub-processes must be defined at design time. MQ Workflow supports time-

outs and, when they occur, will branch to a pre-defined exception path and/or send a message to an administrator. SAP R/3 provides for pre-defined branches which, when an exception occurs, allows an administrator to manually choose one of a set of possible branches.

Among the non-commercial systems, the *OPERA* prototype [9] incorporates language constructs for exception handling and allows for exceptions to be handled at the task level, or propagated up various ancestor levels throughout the running instance. It also removes the need to define the exception handler *a-priori*, although the types of exceptions handled are transactional rather than control flow oriented. The *eFlow* system [28] uses rules to define exceptions, although they cannot be defined separately to the standard model. *ADEPT* [29] supports modification of a process during execution (i.e. add, delete and change the sequence of tasks) both at the type (dynamic evolution) and instance levels (ad-hoc changes). Such changes are made to a traditional monolithic model and must be achieved via manual intervention. The *ADOME* system [30] provides templates that can be used to build a workflow model, and provides some support for (manual) dynamic change. A catalog of 'skeleton' patterns that can be instantiated or specialised at design time is supported by the *WERDE* system [5]. Again, there is no scope for specialisation changes to be made at runtime. *AgentWork* [31] provides the ability to modify process instances by dropping and adding individual tasks based on events and ECA rules. However, the rules do not offer the flexibility or extensibility of Ripple Down Rules, and changes are limited to individual tasks, rather than the task-process-specification hierarchy supported by the Worklet Service. Also, the possibility exists for conflicting rules to generate incompatible actions, which requires manual intervention and resolution.

It should be noted that only a small number of academic prototypes have had any impact on the frameworks offered by commercial systems [32]. Nevertheless, there are some interesting commercial products that offer innovative features with respect to handling exceptions, for example *FLOWer* supports the concept of case-handling; the process model only describes the preferred way of doing things and a variety of mechanisms are offered to allow users to deviate in a controlled manner [1].

The implementation discussed in this paper differs considerably from the above approaches. Exlets, that may include worklets as compensatory processes, dynamically linked to extensible Ripple Down Rules, provide an novel alternative method for the provision of dynamic flexibility and exception handling in workflows.

8 Conclusion and Future Work

Workflow management systems impose a certain rigidity on process definition and enactment because they generally use frameworks based on assembly line metaphors rather than on ways work is actually planned and carried out. An analysis of Activity Theory provided principles of work practices that were used

as a template on which a workflow service has been built that better supports flexibility and dynamic evolution through innovative exception handling techniques. By capturing contextual data, a repertoire of actions is constructed that allow for contextual choices to be made from the repertoire at runtime to efficiently carry out work tasks. These actions, whether exlets or worklets, directly provide for process evolution, flexibility and dynamic exception handling, and mirror accepted work practices.

This implementation presents several key benefits, including:

- A process modeller can describe the standard activities and actions for a workflow process, and any deviations, using the same methodology;
- It allows re-use of existing process components and aids in the development of fault tolerant workflows using pre-existing building blocks [7];
- Its modularity simplifies the logic and verification of the standard model, since individual worklets are less complex to build and therefore easier to verify than monolithic models;
- It provides for a variety of workflow views of differing granularity, which offers ease of comprehensibility for all stakeholders;
- It allows for gradual and ongoing evolution of the model, so that global modification each time a business practice changes or a deviation occurs is unnecessary; and
- In the occurrence of an unexpected event, the process modeller needs simply to choose an existing exlet or build a new one for that event, which can be automatically added to the repertoire for current and future use as necessary, thus avoiding manifold complexities including downtime, model restructuring, versioning problems and so on.

This implementation uses the open-source, service-oriented architecture of YAWL to develop a service for dynamic exception handling completely independent to the core engine. Thus, the implementation may be viewed as a successful case study in service-oriented computing. As such, the approach and resultant software can also be used in the context of other process engines (for example BPEL based systems, classical workflow systems, and the Windows Workflow Foundation). One of the more interesting things to be incorporated in future work is the application of process mining techniques to the various logs collected by the Worklet service; a better understanding of when and why people tend to "deviate" from a work plan is essential for providing better tool support. Archival and resource data will also be useful for refining the contextual choices defined in the rule set.

All system files, source code and documentation for YAWL and the worklet service, including the examples discussed in this paper, may be downloaded via `www.yawl-system.com`.

References

1. W.M.P. van der Aalst, Mathias Weske, and Dolf Grünbauer. Case handling: A new paradigm for business process support. *Data & Knowledge Engineering*, 53(2):129–162, 2005.
2. Gregor Joeris. Defining flexible workflow execution behaviors. In Peter Dadam and Manfred Reichert, editors, *Enterprise-wide and Cross-enterprise Workflow Management: Concepts, Systems, Applications*, volume 24 of *CEUR Workshop Proceedings*, pages 49–55, Paderborn, Germany, October 1999.
3. Alex Borgida and Takahiro Murata. Tolerating exceptions in workflows: a unified framework for data and processes. In *Proceedings of the International Joint Conference on Work Activities, Coordination and Collaboration (WACC'99)*, pages 59–68, San Francisco, CA, February 1999. ACM Press.
4. S. Rinderle, M. Reichert, and P. Dadam. Correctness criteria for dynamic changes in workflow systems: A survey. *Data and Knowledge Engineering*, 50(1):9–34, 2004.
5. Fabio Casati. A discussion on approaches to handling exceptions in workflows. In *CSCW Workshop on Adaptive Workflow Systems*, Seattle, USA, November 1998.
6. C.A. Ellis, K. Keddara, and G. Rozenberg. Dynamic change within workflow systems. In N. Comstock, C. Ellis, R. Kling, J. Mylopoulos, and S. Kaplan, editors, *Proceedings of the Conference on Organizational Computing Systems*, pages 10–21, Milpitas, California, August 1995. ACM SIGOIS, ACM Press, New York.
7. Claus Hagen and Gustavo Alonso. Flexible exception handling in process support systems. Technical report no. 290, ETH Zurich, 1998.
8. Fabio Casati, MariaGrazia Fugini, and Isabelle Mirbel. An environment for designing exceptions in workflows. *Information Systems*, 24(3):255–273, 1999.
9. Claus Hagen and Gustavo Alonso. Exception handling in workflow management systems. *IEEE Transactions on Software Engineering*, 26(10):943–958, October 2000.
10. Mark S. Ackerman and Christine Halverson. Considering an organization's memory. In *Proceedings of the ACM 1998 Conference on Computer Supported Cooperative Work*, pages 39–48. ACM Press, 1998.
11. Peter A. K. Larkin and Edward Gould. Activity theory applied to the corporate memory loss problem. In L. Svennson, U. Snis, C. Sorensen, H. Fagerlind, T. Lindroth, M. Magnusson, and C. Ostlund, editors, *Proceedings of IRIS 23 Laboratorium for Interaction Technology*, University of Trollhattan Uddevalla, 2000.
12. Jakob E. Bardram. I love the system - I just don't use it! In *Proceedings of the 1997 International Conference on Supporting Group Work (GROUP'97)*, Phoenix, Arizona, 1997.
13. I. Bider. Masking flexibility behind rigidity: Notes on how much flexibility people are willing to cope with. In J. Castro and E. Teniente, editors, *Proceedings of the CAiSE'05 Workshops*, volume 1, pages 7–18, Porto, Portugal, 2005. FEUP Edicoes.
14. Michael Adams, Arthur H. M. ter Hofstede, David Edmond, and W.M.P. van der Aalst. Worklets: A service-oriented implementation of dynamic flexibility in workflows. In R. Meersman and Z. Tari et. al., editors, *Proceedings of the 14th International Conference on Cooperative Information Systems (CoopIS'06)*, volume LNCS 4275, pages 291–308, Montpellier, France, November 2006. Springer-Verlag.
15. Michael Adams, Arthur H. M. ter Hofstede, David Edmond, and W.M.P. van der Aalst. Facilitating flexibility and dynamic exception handling in workflows through worklets. In Orlando Bello, Johann Eder, Oscar Pastor, and João Falcão e Cunha, editors, *Proceedings of the CAiSE'05 Forum*, pages 45–50, Porto, Portugal, June 2005. FEUP Edicoes.

16. N. Russell, W.M.P. van der Aalst, and A.H.M. ter Hofstede. Workflow exception patterns. In Eric Dubois and Klaus Pohl, editors, *Proceedings of the 18th International Conference on Advanced Information Systems Engineering (CAiSE 2006)*, pages 288–302, Luxembourg, June 2006. Springer.
17. W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005.
18. W.M.P. van der Aalst, L. Aldred, M. Dumas, and A.H.M. ter Hofstede. Design and implementation of the YAWL system. In A. Persson and J. Stirna, editors, *Proceedings of The 16th International Conference on Advanced Information Systems Engineering (CAiSE 04)*, volume 3084 of *LNCS*, pages 142–159, Riga, Latvia, June 2004. Springer Verlag.
19. Michael Adams, David Edmond, and Arthur H.M. ter Hofstede. The application of activity theory to dynamic workflow adaptation issues. In *Proceedings of the 2003 Pacific Asia Conference on Information Systems (PACIS 2003)*, pages 1836–1852, Adelaide, Australia, July 2003.
20. Yrjo Engestrom, Reijo Miettinen, and Raija-Leena Punamaki, editors. *Perspectives on Activity Theory*. Cambridge University Press, 1999.
21. Y. Engestrom. *Learning by Expanding: An Activity-Theoretical Approach to Developmental Research*. Orienta-Konsultit, Helsinki, 1987.
22. Bonnie A. Nardi, editor. *Context and Consciousness: Activity Theory and Human-Computer Interaction*. MIT Press, Cambridge, Massachusetts, 1996.
23. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(3):5–51, July 2003.
24. P. Compton and B. Jansen. Knowledge in context: A strategy for expert system maintenance. In J.Siekmann, editor, *Proceedings of the 2nd Australian Joint Artificial Intelligence Conference*, volume 406 of *Lecture Notes in Artificial Intelligence*, pages 292–306, Adelaide, Australia, November 1988. Springer-Verlag.
25. Tobias Scheffer. Algebraic foundation and improved methods of induction of ripple down rules. In *Proceedings of the Pacific Rim Workshop on Knowledge Acquisition*, pages 279–292, Sydney, Australia, 1996.
26. B. Drake and G. Beydoun. Predicate logic-based incremental knowledge acquisition. In P. Compton, A. Hoffmann, H. Motoda, and T. Yamaguchi, editors, *Proceedings of the sixth Pacific International Knowledge Acquisition Workshop*, pages 71–88, Sydney, December 2000.
27. Fabio Casati and Giuseppe Pozzi. Modelling exceptional behaviours in commercial workflow management systems. In *1999 IFCIS International Conference on Cooperative Information Systems*, pages 127–138, Edinburgh, Scotland, 1999.
28. Fabio Casati, Ski Ilnicki, LiJie Jin, Vasudev Krishnamoorthy, and Ming-Chien Shan. Adaptive and dynamic composition in eFlow. In *12th International Conference, CAiSE 2000*, pages 13–31, Stockholm, Sweden, 2000.
29. Clemens Hensinger, Manfred Reichert, Thomas Bauer, Thomas Strzeletz, and Peter Dadam. $\text{ADEPT}_{workflow}$ - advanced workflow technology for the efficient support of adaptive, enterprise-wide processes. In *Conference on Extending Database Technology*, pages 29–30, Konstanz, Germany, March 2000.
30. Dickson Chiu, Qing Li, and Kamalakar Karlapalem. A logical framework for exception handling in ADOME workflow management system. In *12th International Conference CAiSE 2000*, pages 110–125, Stockholm, Sweden, 2000.
31. Robert Muller, Ulrike Greiner, and Erhard Rahm. AgentWork: a workflow system supporting rule-based workflow adaptation. *Data & Knowledge Engineering*, 51(2):223–256, November 2004.

32. Michael zur Muehlen. *Workflow-based Process Controlling. Foundation, Design, and Implementation of Workflow-driven Process Information Systems*, volume 6 of *Advances in Information Systems and Management Science*. Logos, Berlin, 2004.