Dynamic, extensible and context-aware exception handling for workflows

Dynamic, Extensible and Context-Aware Exception Handling for Workflows

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

[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 resultant service implements a detailed taxonomy of workflow exception patterns to provide an extensible repertoire of self-contained exception-handling processes called *exlets*, which may be applied at the task, case or specification levels. When an exception occurs at runtime, an exlet is dynamically selected from the repertoire depending on the context of the exception and of the particular work instance. Both expected and unexpected exceptions are catered for in real time, so that 'manual handling' is avoided.

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 [1,2,3]. 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 [4]; and (ii) process exceptions, or deviations from the prescribed process model at runtime [5,6].

For exceptions, the accepted practice is that if it can conceivably be anticipated, then it should be included in the static process model. However, this approach can lead to very complex models, much of which will not be executed in most instances. Also, mixing business logic with exception handling routines complicates the verification and modification of both [7], in addition to rendering the process model almost unintelligible to many stakeholders.

Conversely, if an *unexpected* exception occurs, then the model is deemed to be simply deficient and thus must be amended to include the previously unimagined event (see for example [8]), which disregards the frequency of such events and the costs involved with their correction. Most often, the only available options

R. Meersman and Z. Tari et al. (Eds.): OTM 2007, Part I, LNCS 4803, pp. 95–112, 2007.

are suspension while the exception is handled manually or termination of the case, but since most processes are long and complex, neither option presents a satisfactory solution [7]. Manual handling incurs an added penalty: the corrective actions undertaken are not added to 'organisational memory' [9,10], and so natural process evolution is not incorporated into future iterations of the process. Associated problems include those of migration, synchronisation and version control [5].

Thus a large group of business processes do not easily map to the rigid modelling structures provided [11], due to the lack of flexibility inherent in a framework that, by definition, imposes rigidity. Business processes are 'system-centric', or *straight-jacketed* [2] into the supplied framework, rather than truly reflecting the way work is actually performed [1]. As a result, users are forced to work outside of the system, and/or constantly revise the static process model, in order to successfully perform their activities, thereby negating the efficiency gains sought by implementing a workflow solution in the first place.

The flux inherent in work practices has been further borne out by our previous work on process mining. When considering processes where people are expected to execute tasks in a structured way but are not forced to by a workflow system, process mining shows that the processes are much more dynamic than expected. That is, workers tend to deviate from the 'normal flow', often with good reasons.

To gain a grounded understanding of actual work practices, we previously undertook a detailed study of *Activity Theory*, a broad collective of theorising and research in organised human activity (cf. [12,13]), and derived from it a set of principles that describe the nature of participation in organisational work practices [14]. We then applied those principles to the design and implementation of a discrete web-based service that maintains an extensible repertoire of self-contained exception handling processes, known as *exlets*, and an associated set of contextual selection rules, to dynamically support exception handling for business process instances orthogonal to the underlying workflow engine.

This paper describes the resultant service, which uses the *'worklets'* approach introduced in [15,16] as a conceptual foundation, and applies the classification of workflow exception patterns from [17]. The implementation platform used is the well-known workflow environment YAWL [18,19], which supports a Service Oriented Architecture (SOA) — but the discrete nature of the service means its applicability is in no way limited to the YAWL environment. Also, being open-source, the service is freely available for use and extension.

The paper is organised as follows: Section 2 provides an overview of the design and operation of the service, while Section 3 details the service architecture. Section 4 discusses exception types handled by the service and the definition of exlets. Section 5 describes how the approach utilises *Ripple Down Rules* (RDR) to achieve contextual, dynamic selection of exlets at runtime. Section 6 discusses related work, and finally Section 7 concludes the paper.

2 Service Overview

The implemented service, known as the *Worklet Service*[1] comprises two distinct but complementary sub-services: a *Selection sub-service*, which enables dynamic flexibility for process instances (cf. [15]); and an *Exception Handling sub-service* (the subject of this paper), which provides facilities to handle both expected and unexpected process exceptions at runtime.

The Exception Handling sub-service (or, more simply, the *Exception Service*) allows administrators to define exception handling processes (called *exlets*) for parent workflow instances, to be invoked when certain events occur, and thereby allowing execution of the parent process to continue unhindered. It has been designed so that the enactment engine, besides providing notifications at certain points in the life cycle of a process instance, needs no knowledge of an exception occurring, nor of any consequent invocation of exlets — all exception checking and handling is provided by the service. Additionally, all exlets in a specification's repertoire 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 any requirement to modify the original process definition.

The Exception Service is built on the same conceptual framework as the Worklet Selection sub-service, and so uses the same repertoire and dynamic rules approach (see Section 5). 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 rule being satisfied is an exlet (which may contain a worklet to be executed as a compensation process). Second, while the Selection Service is invoked for certain nominated tasks in a process, the Exception Service, when enabled, is invoked for *every* case and task executed by the enactment engine, and will detect and handle up to ten different kinds of process exceptions (those exception types are described in Section 4.1).

Most modern programming languages provide mechanisms that separate exception handling routines from the 'normal' program logic, which facilitates the design of readable, comprehensible programs [20,21,22]. Similar methods are incorporated into distributed frameworks and operating systems. However, little or no such means are provided in most WfMSs. Usually, any or all possible exceptions must be incorporated into the monolithic workflow model, which contravenes accepted paradigms of modularity, encapsulation and reusability.

For the Exception Service, an exlet (discrete and external to the parent model) may consist of a number of various actions (such as cancel, suspend, complete, fail, restart and compensate) and be automatically applied at a workitem, case and/or specification level. And, because exlets can include worklets as compensation processes, the original parent process model only needs to reveal the actual business logic for the process.

[1] Essentially, a *worklet* is a small, discrete workflow process that may act as both a late-bound sub-net for an enabled workitem and a compensation process within an exception handler.

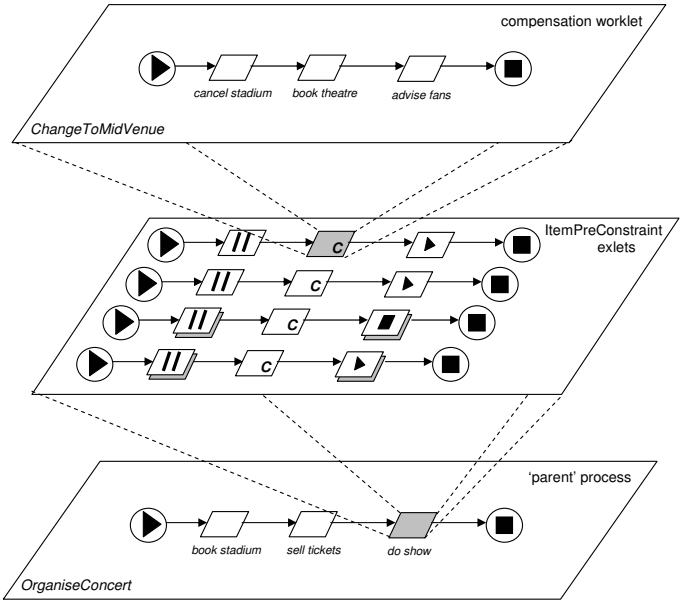

Fig. 1. Process – Exlet – Worklet Hierarchy

Each time an exception occurs, the service makes a choice from the repertoire based on the type of exception and the contextual data of the workitem/case, using a set of rules to select the most appropriate exlet to execute (see Section 5). If the exlet contains a compensation primitive, the associated worklet 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 execution logs that may be combined with those of the engine 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 as an example a simple process for the organisation of a rock concert (*Organise Concert*).

The repertoire of exlets grows as new exceptions arise or different ways of handling exceptions are formulated, *including while the parent process is executing*, and those handling methods automatically become an implicit part of the process specification for all current and future instances of the process.

Any number of exlets can form the repertoire of each particular exception type for 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 Selection and Exception sub-services can be used in combination within case instances to achieve dynamic flexibility *and* exception handling simultaneously.

3 Service Architecture

The Worklet Service has been implemented as a YAWL Custom Service [18,19]. 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 based systems, classical workflow systems, and the Windows Workflow Foundation) by making the necessary linkages in the service interface. As such, this implementation also represents a case study in service-oriented computing whereby dynamic flexibility and exception handling for workflows, orthogonal to the underlying workflow language, is provided.

To enable the Worklet Service to serve a workflow enactment engine, a number of events and methods must be provided by an interface between them. In the conceptualisation and design of the service, the size or 'footprint' of the interface has been kept to an absolute minimum to accommodate ease-of-deployment and thus maximise the installation base, or the number of enactment engines, that may benefit from the extended capabilities that the worklet service offers. Being a web-service, the worklet service has been designed to enable remote deployment and to allow a single instance of the service to concurrently manage the flexibility and exception handling management needs for a number of disparate enactment engines that conform to the interface.

The YAWL environment provides for the workflow enactment engine and external services to interact across several interfaces supporting the ability to send and receive both messages and XML data to and from the engine. Three of those interfaces are used by the Worklet Exception Service:

- *Interface A* provides endpoints for process definition, administration and monitoring [19] – the service uses Interface A to upload worklet specifications to the engine;
- *Interface B* provides endpoints for client and invoked applications and workflow interoperability [19] – 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 'eXception') which has been designed to provide the engine with the ability to notify custom services of certain events and checkpoints during the life-cycle of each process instance and each of its tasks where process exceptions either may have occurred or should be tested for. Thus Interface X provides the Exception Service with the necessary triggers to dynamically capture and handle process exceptions.

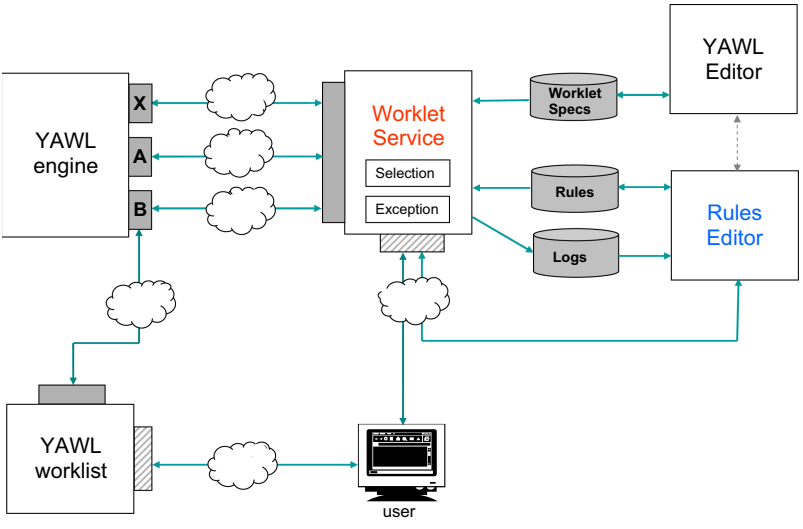

Fig. 2. External Architecture of the Worklet Service

In fact, Interface X was created to enable the Exception 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. For example, in addition to exception handling, the interface's methods provide the tools to enable ad-hoc or permanent adaptations to process schemas, such as re-doing, skipping, replacing and looping of tasks.

Figure 2 shows the 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 5). The service also provides servlet pages that allow users to directly communicate with the service to raise external exceptions and carry out administration tasks.

4 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 Exception Service. It then describes the handling primitives that may be used to form an exception handling process (i.e. an exlet). 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 [17].

4.1 Exception Types

The following seven types of exceptions are supported by our current implementation:

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.

When the service receives an constraint event notification, the rule set is queried (see Section 5), and if a constraint has been violated the associated exlet is selected and invoked.

TimeOut: A timeout event occurs when a workitem reaches a set deadline. The service receives a reference to the workitem and to each of the other workitems running in parallel to 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 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 directly by a user via a servlet page (for example, Figure 5); 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).

Three more exception types have been identified but are not yet supported, since they rely more heavily on the internal mechanisms of the enactment engine:

ItemAbort: This 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).

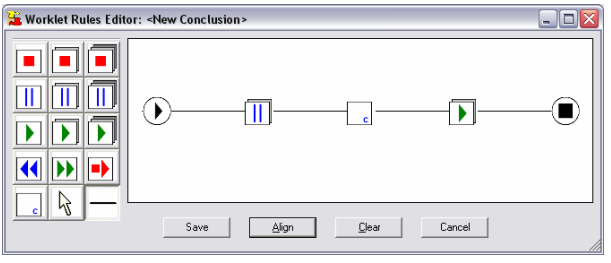

Fig. 3. Example Exlet in the Rules Editor

4.2 Exception Handling Primitives

Each exlet is defined graphically using the Worklet Rules Editor, and may contain any number of steps, or *primitives*. Figure 3 shows the Rules Editor with an example exlet displayed. On the left of the Editor is the set of available primitives, which are (reading left-to-right, top-to-bottom):

- *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.
- *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 the actions of previous primitives in the exlet, the worklet may execute simultaneously to the parent case, or execute while the parent is suspended. One or more worklets may be simultaneously invoked by a compensate primitive.

Thus, the example exlet in Figure 3 will suspend the case, execute a compensation process, then continue (or unsuspend) the case. A compensation primitive may contain an array of one or more worklets – when multiple worklets are defined they are launched concurrently as an effectively composite compensatory action. Execution moves to the next primitive in the exlet when all worklets have completed. Additionally, relevant data values may be mapped from a case to a compensatory worklet, where they may be modified and mapped back again to the original case.

Worklets that are executed as compensatory processes within exlets can in turn invoke child worklets to any depth. The primitives 'Suspend All Cases', 'Continue All Cases' and 'Remove All Cases' may be flagged 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 another worklet and/or an exlet, and so on. Since compensatory worklets are launched as separate cases in the enactment engine, they too are monitored by the service for exceptions and thus may have exlets launched for them in certain circumstances. Also, the 'Continue' primitives are applied only to those workitems and cases that were previously suspended by the same exlet. Execution moves to the next primitive in the exlet when all worklets launched from a compensation primitive have completed.

Referring to Figure 1, the centre tier shows the exlets repertoire for an ItemPreConstraint violation for a particular task, which correspond to the rule tree shown in Figure 4. There may actually be up to eleven different 'planes' for this tier — one for each exception type. 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.

5 Contextual Selection of Exlets

The runtime selection of an exlet relies on the type of exception that has occurred and the relevant context of the workitem and/or case instance, derived from data attribute values of the case instance, workitem-level values, the internal status of each workitem in the process instance, resource data, historical data

from process logs, and other extensible external sources. Some of these data are supplied directly by the enactment engine across the interfaces, others may be indirectly supplied using process mining techniques.

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].

Each specification may have an associated rule set, which consists of a set 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 Figure 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. For non-terminal nodes, if a node's condition evaluates to *True*, the node connected on its *True* branch is subsequently evaluated; if it evaluates to *False*, the node connected on its *False* branch 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 conclusion of 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 is an exception rule of the more general one of its parent (that is, it is a *refinement* of the more general 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. The conclusion of a node equates to an exlet.
- **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 exception type. The tree set of a case-level exception type will contain exactly one tree. The tree set of an item-level type will contain one rule tree for each task of the specification that has rules defined for it.
- **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 exception type for which rules have been defined.

A repertoire of exlets may be formed for each exception type. Each specification has a unique rule set (if any), which contains between one and eleven tree

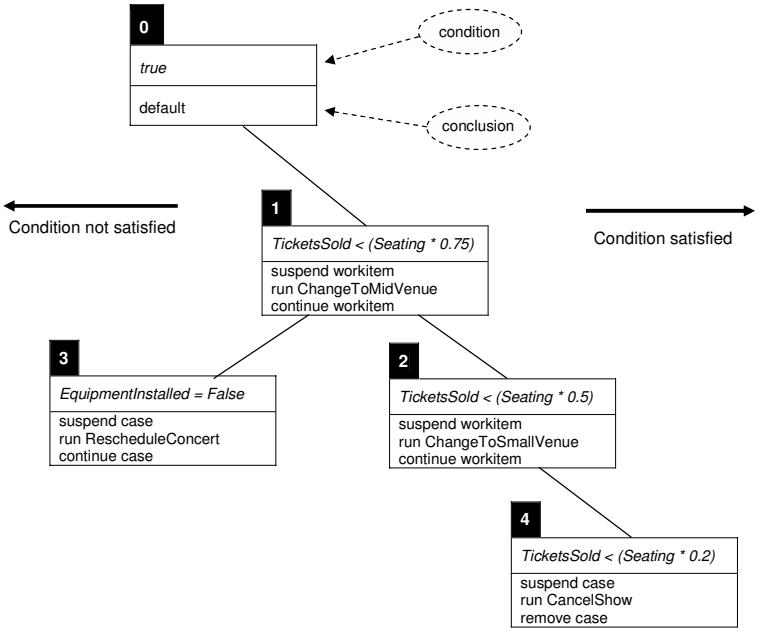

Fig. 4. Example rule tree (ItemPreConstraint for DoShow task of OrganiseConcert)

sets (or sets of rule trees), one for selection rules (used by the Selection subservice) 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. The rule set for each specification is stored as XML data in a discrete disk file. All rule set files are stored in the worklet repository.

If there are no rules defined for a certain exception type in the rule set for a specification, a runtime event of that type is 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. So, for example, if an administrator 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 (that is, rules defined within those tree sets). Of course, rules for other types can be added later when required. Figure 4 shows the ItemPreConstraint rule tree for the third task in the Organise Concert example, *Do Show*, (corresponding to the centre and lower tiers of Figure 1 respectively); it is evaluated when a *Do Show* workitem instance is enabled. This rule tree provides exlets for organisers to change the venue of the concert, or cancel it, when there are insufficient tickets sold to fill the original venue. For example, if a particular *Do Show* instance has

Fig. 5. Raise Case-Level Exception Screen (Organise Concert example)

a value for the attribute 'TicketsSold' that 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 unsuspends) the workitem. By following the exception path of the rule tree, it can be seen that each subsequent node is a refinement of its parent, since it is only evaluated if its parent rule is satisfied. So, 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[2].

As mentioned previously, the service provides a set of servlet pages that 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) at any time during the execution of a case. When invoked, the service lists from the rule set for the selected case the existing external exception triggers (if any) for the case's specification (see Figure 5). Note that these triggers may 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). When a trigger is selected by the

[2] It has been formally shown that an RDR tree traverses through a smaller number of rules enroute to its final conclusion than traversal through an equivalent decision list [25].

user, the corresponding rule set is queried and the appropriate exlet, relative to the case's context and the trigger selected, is executed. Item-level external exceptions can be raised in a similar way.

Notice that at the bottom of the list of triggers in Figure 5 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, via another servlet page, of the new exception, its context and possible ways to handle it — the notification of an unexpected external exception automatically suspends the case as a safeguard. 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. New exlets for unexpected internal exceptions are raised and launched using the same approach as that described for the Selection sub-service in [15].

The examples used in this section have been intentionally simplified to demonstrate the operation of the Exception Service; while not intended to portray a realistic process, it is desirable to not camouflage the subject of this paper by using a more realistic, and thus a necessarily more complex process. However, exemplary studies have been undertaken using real-world processes from both a relatively rigid business scenario and a more creative environment, which serve to fully validate the approach (see Chapter 7 of [27]).

6 Related Work

Since the mid-nineties much research has been carried out on issues related to exception handling in workflow management systems. Such research was initiated because, generally, commercial workflow management systems provide only basic support for handling exceptions [17,28,7,29] (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.

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. For a more systematic overview see [17], where different tools are evaluated with respect to their exception handing capabilities using a patterns-based approach.

Tibco iProcess 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. *Websphere MQ Workflow* supports timeouts and, when they occur, will branch to a pre-defined exception path and/or send a message to an administrator. SAP Workflow supports exception events for cancelling workflow instances, for

checking workitem pre- and post- constraints, and for 'waiting' until an external trigger occurs. Exception handling processes may be assigned to a workflow based on the type of exception that has occurred, although the handlers for each are specified at design time, and only one may be assigned to each type. FLOWer is described as a 'case-handling' system, and supports some exception handling actions [2]. For example, a deadline expiry can automatically complete a workitem. Also, some support for constraint violation is offered: a plan may be automatically created or completed when a specified condition evaluates to true [17].

Among the non-commercial systems, the *OPERA* prototype [7] has a modular structure in which activities are nested. When a task fails, its execution is stopped and the control of the process is handed over to a single handler predefined for that type of exception — the context of the activity is not accounted for. If the handler cannot solve the problem, it propagates the exception up the activity tree; if no handler can be found the entire process instance aborts. The *eFlow* system [30] supports the definition of compensation rules for regions, although they are static and cannot be defined separately to the standard model. *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. The *TREX* system [32] allows the modeller to choose from a catalog of exception handlers during runtime to handle exceptions as they arise; however, it requires a human agent to intervene whenever an exception occurs. Also, the exceptions handled are, for the most part, transactional, and scope for most expected and all unexpected exceptions is not provided. The *MARIFlow* system [33] supports document exchange and coordination across the internet. The system supports some transactional-level exception handling, for example rolling back and restarting a blocked or dead process, but there is no support for dynamic change or handling exceptions within the control flow of the process. *CBRFlow* [34] uses a case-based reasoning approach to support adaptation of predefined workflow models to changing circumstances by allowing (manual) annotation of business rules during run-time via incremental evaluation. It should be noted that only a small number of academic prototypes have had any impact on the frameworks offered by commercial systems [17,28].

The majority of languages used to described and define business process models are of a procedural nature, which limits their effectiveness in very flexible environments [35]. For example, BPEL provides *compensation handlers* that are intended to support rollback or undo of activities after an error has occurred; however, they are essentially unable to access the current process state [36] — thus the context of the case cannot be taken into account. In addition, compensation handlers cannot affect other process instances (i.e. at the specification level), cannot be used to effect non-erroneous changes in process execution [36]

and may only perform a termination action (all in contrast to the various actions supported by the Worklet Exception Service). Also, BPEL offers no support for constraint violations [17].

In summary, approaches to workflow flexibility and exception handling usually rely on a high-level of runtime user interactivity, which directly impedes on the basic aim of workflow systems (to bring greater efficiencies to work practices) and distracts users from their primary work procedures into process support activities. Another common theme is the complex update, modification and migration issues required to evolve process models. The Worklet Service differs considerably from those approaches. Exlets, that may include worklets as compensatory processes, as members of a repertoire, and dynamically linked to Ripple Down Rule sets, provide a novel, complete and extensible approach for the provision of dynamic exception handling in workflows.

7 Conclusion

The Worklet Exception Service has been constructed around the idea that exceptions, or deviations from a process specification, are a natural occurrence within almost every instantiation of a process. Thus the service provides a fully featured exception handling paradigm that detects (through constraint checking), reacts to, handles and incorporates exceptions and the way they are handled as they occur. The service also allows for unexpected exceptions to be handled during execution, so that a process instance need not be terminated when one occurs, or be handled off-system.

The service provides easy to use mechanisms to incorporate new handling procedures for unexpected exceptions implicitly into the process specification so that they are automatically available for all current and future instantiations of the specification. Thus a repertoire of exception handling procedures is maintained by the service for each process specification, so completely avoiding the need to modify a specification each time a deviation from its prescribed flow occurs — which also avoids the on-costs associated with taking the specification off-line while modifications are performed and verified, versioning problems, migration control issues and so on.

In providing these benefits, the Worklet Exception Service:

- Keeps the parent model clean and relatively simple;
- Promotes the reuse of sub-processes in different models, and allows standard processes to also be used as compensation processes, and vice versa;
- Maintains an extensible repertoire of exlets that can be constructed during design and/or runtime and can be invoked as required;
- Allows a specification to implicitly build a history of executions, providing for a learning system that can take the appropriate actions within certain contexts automatically;
- Maintains exlets, and compensatory worklets, as fully encapsulated processes, which allows for easier verification and modification; and

– Allows a model to evolve without the need to stop and modify the design of the whole specification when an exception occurs.

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. Bider, I.: Masking flexibility behind rigidity: Notes on how much flexibility people are willing to cope with. In: Castro, J., Teniente, E. (eds.) CAiSE 2005 Workshops, vol. 1, pp. 7–18, FEUP Edicoes, Porto (2005)
2. van der Aalst, W.M.P., Weske, M., Grünbauer, D.: Case handling: A new paradigm for business process support. Data & Knowledge Engineering 53(2), 129–162 (2005)
3. Joeris, G.: Defining flexible workflow execution behaviors. In: Dadam, P., Reichert, M. (eds.) Enterprise-wide and Cross-enterprise Workflow Management: Concepts, Systems, Applications. CEUR Workshop Proceedings, Paderborn, Germany, vol. 24, pp. 49–55 (October 1999)
4. Borgida, A., Murata, T.: Tolerating exceptions in workflows: a unified framework for data and processes. In: WACC 1999. Proceedings of the International Joint Conference on Work Activities, Coordination and Collaboration, pp. 59–68. ACM Press, San Francisco, California, USA (1999)
5. Rinderle, S., Reichert, M., Dadam, P.: Correctness criteria for dynamic changes in workflow systems: a survey. Data and Knowledge Engineering 50(1), 9–34 (2004)
6. Casati, F.: A discussion on approaches to handling exceptions in workflows. In: Proceedings of the CSCW Workshop on Adaptive Workflow Systems, Seattle, USA (November 1998)
7. Hagen, C., Alonso, G.: Exception handling in workflow management systems. IEEE Transactions on Software Engineering 26(10), 943–958 (2000)
8. Casati, F., Fugini, M., Mirbel, I.: An environment for designing exceptions in workflows. Information Systems 24(3), 255–273 (1999)
9. Ackerman, M.S., Halverson, C.: Considering an organization's memory. In: Proceedings of the ACM 1998 Conference on Computer Supported Cooperative Work, pp. 39–48. ACM Press, Seattle, Washington, USA (1998)
10. Larkin, P.A.K., Gould, E.: Activity theory applied to the corporate memory loss problem. In: Svennson, L., Snis, U., Sorensen, C., Fagerlind, H., Lindroth, T., Magnusson, M., Ostlund, C. (eds.) Proceedings of IRIS 23 Laboratorium for Interaction Technology, University of Trollhattan Uddevalla, Sweden (2000)
11. Bardram, J.E.: I love the system - I just don't use it! In: Jakob, E. (ed.) GROUP 1997. Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work, Phoenix, Arizona, USA, pp. 251–260. ACM Press, New York (1997)
12. Engestrom, Y., Miettinen, R., Punamaki, R.-L. (eds.): Perspectives on Activity Theory. Cambridge University Press, Cambridge (1999)
13. Nardi, B.A. (ed.): Context and Consciousness: Activity Theory and Human-Computer Interaction. MIT Press, Cambridge, Massachusetts (1996)
14. Adams, M., Edmond, D., ter Hofstede, A.H.M.: The application of activity theory to dynamic workflow adaptation issues. In: PACIS 2003. Proceedings of the 2003 Pacific Asia Conference on Information Systems, Adelaide, Australia, pp. 1836–1852 (July 2003)

15. Adams, M., ter Hofstede, A.H.M., Edmond, D., van der Aalst, W.M.P.: Worklets: A service-oriented implementation of dynamic flexibility in workflows. In: Meersman, R., Tari, Z. (eds.) On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE. LNCS, vol. 4275, pp. 291–308. Springer, Heidelberg (2006)
16. Adams, M., ter Hofstede, A.H.M., Edmond, D., van der Aalst, W.M.P.: Facilitating flexibility and dynamic exception handling in workflows through worklets. In: Bello, O., Eder, J., Pastor, O., Cunha, J.F. (eds.) CAiSE 2005 Forum, pp. 45–50, FEUP Edicoes, Porto (2005)
17. Russell, N., van der Aalst, W.M.P., ter Hofstede, A.H.M.: Workflow exception patterns. In: Dubois, E., Pohl, K. (eds.) CAiSE 2006. LNCS, vol. 4001, pp. 288–302. Springer, Heidelberg (2006)
18. van der Aalst, W.M.P., ter Hofstede, A.H.M.: YAWL: Yet Another Workflow Language. Information Systems 30(4), 245–275 (2005)
19. van der Aalst, W.M.P., Aldred, L., Dumas, M., ter Hofstede, A.H.M.: Design and implementation of the YAWL system. In: Persson, A., Stirna, J. (eds.) CAiSE 2004. LNCS, vol. 3084, pp. 142–159. Springer, Heidelberg (2004)
20. Hagen, C., Alonso, G.: Flexible exception handling in process support systems. Technical report No. 290, ETH Zurich, Switzerland (1998)
21. Lei, Y., Singh, M.P.: A comparison of workflow metamodels. In: Proceedings of the ER-97 Workshop on Behavioral Modeling and Design Transformations: Issues and Opportunities in Conceptual Modeling, Los Angeles, California, USA (November 1997)
22. Goodenough, J.B.: Exception handling: issues and a proposed notation. Communications of the ACM 18(12), 683–696 (1975)
23. van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distributed and Parallel Databases 14(3), 5–51 (2003)
24. Compton, P., Jansen, B.: Knowledge in context: A strategy for expert system maintenance. In: Barter, C.J., Brooks, M.J. (eds.) AI 1988. LNCS, vol. 406, pp. 292–306. Springer, Heidelberg (1990)
25. Scheffer, T.: Algebraic foundation and improved methods of induction of ripple down rules. In: Proceedings of the 2nd Pacific Rim Workshop on Knowledge Acquisition, Sydney, Australia, pp. 279–292 (1996)
26. Drake, B., Beydoun, G.: Predicate logic-based incremental knowledge acquisition. In: Compton, P., Hoffmann, A., Motoda, H., Yamaguchi, T. (eds.) Proceedings of the sixth Pacific International Knowledge Acquisition Workshop, Sydney, Australia, pp. 71–88 (December 2000)
27. Adams, M.: Facilitating Dynamic Flexibility and Exception Handling for Workflows. Phd thesis. Faculty of Information Technology, Queensland University of Technology, Brisbane, Australia (2007), `http://yawlfoundation.org/documents/AdamsWorkletsFinalThesis.pdf`
28. zur Muehlen, M.: Workflow-based Process Controlling. Foundation, Design, and Implementation of Workflow-driven Process Information Systems. In: Advances in Information Systems and Management Science. vol. 6, Logos, Berlin (2004)
29. Casati, F., Pozzi, G.: Modelling exceptional behaviours in commercial workflow management systems. In: CoopIS 1999. Proceedings of the 4th IFCIS International Conference on Cooperative Information Systems, Edinburgh, Scotland, pp. 127–138. IEEE, Los Alamitos (1999)
30. Casati, F., Ilnicki, S., Jin, L., Krishnamoorthy, V., Shan, M.-C.: Adaptive and dynamic composition in eFlow. In: Wangler, B., Bergman, L.D. (eds.) CAiSE 2000. LNCS, vol. 1789, pp. 13–31. Springer, Heidelberg (2000)

31. Muller, R., Greiner, U., Rahm, E.: AgentWork: a workflow system supporting rule-based workflow adaptation. Data & Knowledge Engineering 51(2), 223–256 (2004)
32. van Stiphout, R., Meijler, T.D., Aerts, A., Hammer, D., le Comte, R.: TREX: Workflow TRansactions by Means of EXceptions. Technical report, Eindhoven University of Technology (1997)
33. Dogac, A., Tambag, Y., Tumer, A., Ezbiderli, M., Tatbul, N., Hamali, N., Icdem, C., Beeri, C.: A workflow system through cooperating agents for control and document flow over the internet. In: Scheuermann, P., Etzion, O. (eds.) CoopIS 2000. LNCS, vol. 1901, pp. 138–143. Springer, Heidelberg (2000)
34. Weber, B., Wild, W., Breu, R.: CBRFlow: Enabling adaptive workflow management through conversational case-based reasoning. In: Funk, P., González Calero, P.A. (eds.) ECCBR 2004. LNCS (LNAI), vol. 3155, pp. 434–448. Springer, Heidelberg (2004)
35. Pesic, M., van der Aalst, W.M.P.: A declarative approach for flexible business processes. In: Eder, J., Dustdar, S. (eds.) Business Process Management Workshops. LNCS, vol. 4103, pp. 169–180. Springer, Heidelberg (2006)
36. Coleman, J.W.: Examining BPEL's compensation construct. In: REFT 2005. Proceedings of the Workshop on Rigorous Engineering of Fault-Tolerant Systems, Newcastle upon Tyne, UK, pp. 122–128 (July 2005)