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' is avoided.

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

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) process 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. 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 some stakeholders.

Conversely, if an unexpected exception occurs then the model is deemed to be simply deficient and thus needs to 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 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 [4].

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 to by a 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 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* [1] into the supplied framework, rather than truly reflecting the way work is actually performed [12]. 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.

To better understand actual work practices, we previously undertook a detailed study of *Activity Theory*, a broad collective of theorising and research in organised human activity (cf. [13, 14]) and derived from it a set of principles that describe the nature of participation in organisational work practices [15]. We have applied those principles to the realisation of a discrete service that utilises an extensible repertoire of self-contained exception handling processes and associated selection rules to support the flexible modelling, analysis, enactment and exception handling of business processes for a wide variety of work environments.

This paper introduces the service, which is based on and extends the *'worklets'* approach described in [16] and [17] and applies the classification of workflow exception patterns from [18]. The implementation platform used is the well-known, open-source workflow environment YAWL [19, 20], which supports a Service Oriented Architecture (SOA), and so the service's applicability is in no way limited to the YAWL environment. Also, being open-source, it 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 exception handling processes. Section 5 describes how the approach utilises *Ripple Down Rules* (RDR) to achieve contextual, dynamic selection of handling processes at runtime. Section 6 discusses related work, and finally Section 7 outlines future directions and concludes the paper.

2 Worklet Service Overview

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 process instances; and an *Exception Handling sub-service* (the focus of this paper), which provides facilities to handle both expected and unexpected process exceptions at runtime.

Briefly, the Selection sub-service enables dynamic flexibility by allowing a process designer to designate certain tasks to each be substituted at runtime with a dynamically and contextually selected worklet, which therefore handles one specific task in a larger, composite process. An extensible repertoire of worklets is maintained by the service for each task in a specification. For further information regarding the design philosophy and implementation of the Selection sub-service, see [16].

The Exception sub-service allows process designers to define exception handling processes (called *exlets*) for parent processes to be invoked when exceptions occur and thereby allow the process to continue unhindered. The Exception Handling sub-service uses the same repertoire and dynamic rules approach as the Selection sub-service. There are, however, two fundamental differences between the two sub-services. First, where the Selection sub-service selects a worklet as the result of satisfying a rule in a rule set, the result of an Exception Handling sub-service query is an exlet. Second, while the Selection sub-service is invoked for certain nominated tasks in a process, the Exception Handling sub-service, when enabled, is invoked for *every* case and task executed by the enactment engine. 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 exlet may consist of a number of various actions (such as cancel, suspend, complete, fail, restart and compensate) and be 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, while 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.

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 compensatory worklet 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 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*).

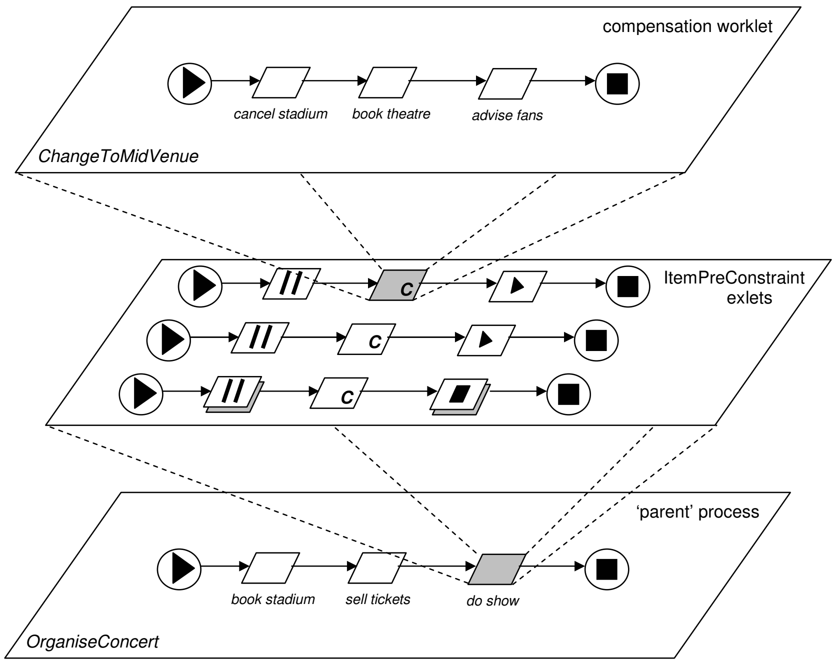


Fig. 1. Process – Exlet – Worklet Hierarchy

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 [19, 20]. 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 [21], 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 flexibility and exception handling for workflows, orthogonal to the underlying workflow language, is provided.

Figure 2 shows the external architecture of the Worklet Service. The YAWL environment allows workflow instances 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 are used by the Worklet Service:

– *Interface A* provides endpoints for process definition, administration and monitoring [20] – used by the service to upload worklet specifications to the engine;
– *Interface B* provides endpoints for client and invoked applications and workflow interoperability [20] – 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') provides endpoints for the engine to notify services of exception events and checkpoints and for services to manage the statuses of tasks and cases.

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

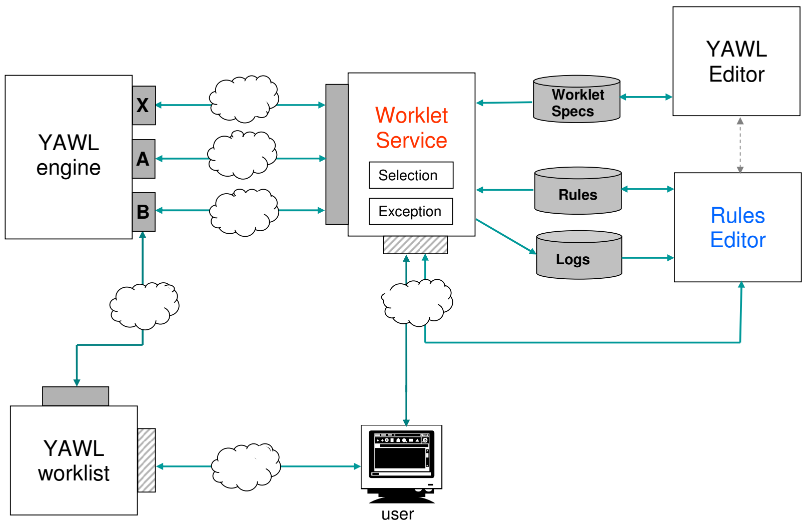


Fig. 2. External Architecture of the Worklet Service

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 Worklet 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 [18].

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 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).

Three more exception types have been identified but are not yet supported:

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).

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.

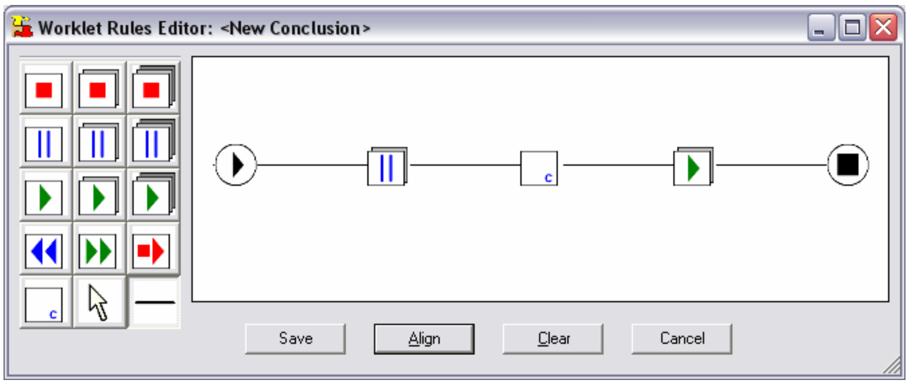


Fig. 3. Example Exlet in the Rules Editor

- *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.

The ‘All Cases’ primitives may be modified in the Rules Editor so that their action is restricted to ancestor cases only – those in a hierarchy of worklets back to the original parent case; an exlet may invoke a compensatory worklet which in turn may invoke an exlet with a compensatory worklet, 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 they are launched concurrently as a composite compensatory action. Execution moves to the next primitive in the exlet when all worklets have completed. Additionally, data values may be mapped from a case to a compensatory worklet and back again.

Referring to Figure 1, the centre tier shows the exlets defined for ItemPreConstraint violations. 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 case instance. 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 [22]. 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 [23].

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 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. For non-terminal nodes, if a node’s condition evaluates to *True*, the node connected on its *True* branch is also evaluated; if it evaluates to *False*, the node connected on its *False* branch is evaluated [24]. 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 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.

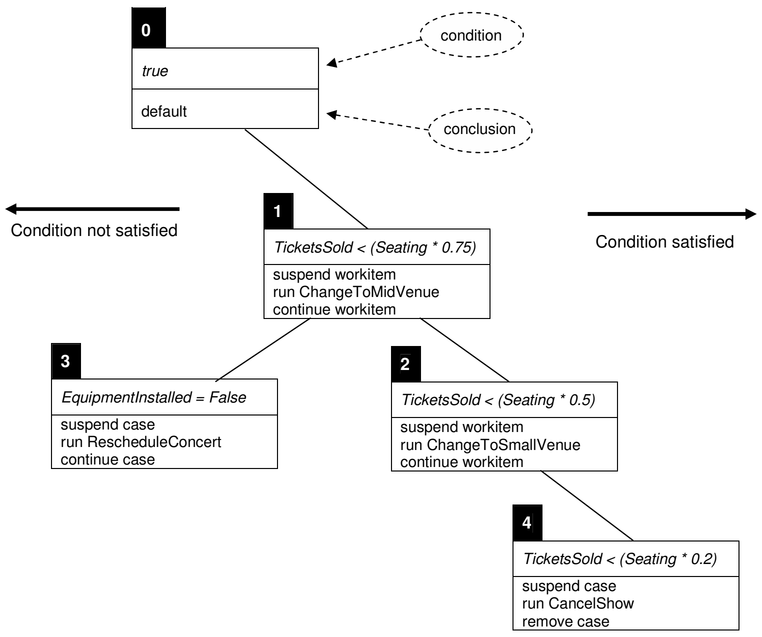


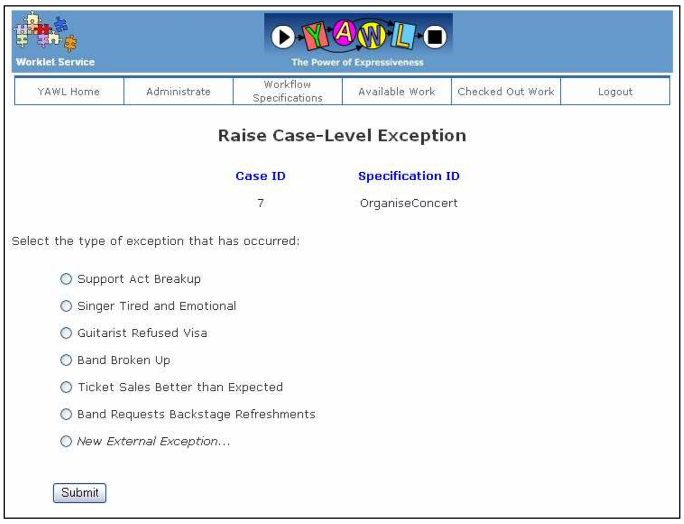
Fig. 4. Example rule tree (for OrganiseConcert ItemPreConstraint)

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 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.

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 of any run-time exception events that aren't defined in the rule set file are simply ignored.

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). 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 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. Following the rule tree, 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.



Raise Case-Level Exception

Case ID	Specification ID
7	OrganiseConcert

Select the type of exception that has occurred:

- ○ Support Act Breakup
- ○ Singer Tired and Emotional
- ○ Guitarist Refused Visa
- ○ Band Broken Up
- ○ Ticket Sales Better than Expected
- ○ Band Requests Backstage Refreshments
- ○ *New External Exception...*

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

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

Notice that at the bottom of the list (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 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, as detailed in [16].

6 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; for a more systematic overview see [18], 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 [7, 25] (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 timeouts 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 [7] 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 [26] uses rules to define exceptions, although they cannot be defined separately to the standard model. *ADEPT* [27] 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 [28] 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* [29] 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 [18, 30].

The Worklet Service 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.

7 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 exlets and worklets is constructed that allow for contextual choices to be made from the repertoire at runtime to efficiently carry out work tasks. These actions directly provide for process evolution, flexibility and dynamic exception handling, and mirror accepted work practices.

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 service; a better understanding of when and why people tend to "deviate" from a work plan is essential for providing better tool support. The application of archival and resource data will also be useful for further 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. Exception handling in workflow management systems. *IEEE Transactions on Software Engineering*, 26(10):943–958, October 2000.
8. Fabio Casati, MariaGrazia Fugini, and Isabelle Mirbel. An environment for designing exceptions in workflows. *Information Systems*, 24(3):255–273, 1999.
9. 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.
10. 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.
11. 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.
12. 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.
13. Yrjo Engestrom, Reijo Miettinen, and Raija-Leena Punamaki, editors. *Perspectives on Activity Theory.* Cambridge University Press, 1999.
14. Bonnie A. Nardi, editor. *Context and Consciousness: Activity Theory and Human-Computer Interaction.* MIT Press, Cambridge, Massachusetts, 1996.
15. 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.
16. 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.
17. 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.
18. 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.
19. W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005.
20. 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.
21. 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.
22. 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.
23. 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.
24. 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.
25. 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.
26. 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.
27. Clemens Hensinger, Manfred Reichert, Thomas Bauer, Thomas Strzeletz, and Peter Dadam. 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.
28. 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.
29. 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.
30. 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.