Facilitating Dynamic Flexibility and Exception Handling for Workflows

by

Michael James Adams

B.Comp (Hons)

A dissertation submitted for the degree of

IF49 Doctor of Philosophy

Principal Supervisor: Associate Professor Arthur H.M. ter Hofstede

Associate Supervisors: Dr David Edmond & Professor Wil M.P. van der Aalst

Faculty of Information Technology

Queensland University of Technology

Brisbane, Australia

23 May 2007

Certificate of Acceptance

Keywords

workflow exception handling, workflow flexibility, adaptive workflow, service oriented architecture, Activity Theory, Ripple-Down Rules, worklet, exlet, YAWL

Abstract

Workflow Management Systems (WfMSs) are used to support the modelling, analysis, and enactment of business processes. The key benefits WfMSs seek to bring to an organisation include improved efficiency, better process control and improved customer service, which are realised by modelling rigidly structured business processes that in turn derive well-defined workflow process instances. However, the proprietary process definition frameworks imposed by WfMSs make it difficult to support (i) dynamic evolution and adaptation (i.e. modifying process definitions during execution) following unexpected or developmental change in the business processes being modelled; and (ii) exceptions, or deviations from the prescribed process model at runtime, even though it has been shown that such deviations are a common occurrence for almost all processes. These limitations imply that a large subset of business processes do not easily translate to the system-centric modelling frameworks imposed.

This research re-examines the fundamental theoretical principles underpinning workflow technologies to derive an approach that moves forward from the production-line paradigm and thereby offers workflow management support for a wider range of work environments. It develops a sound theoretical foundation based on Activity Theory to deliver an implementation of an approach for dynamic and extensible flexibility, evolution and exception handling in workflows, based not on proprietary frameworks, but on accepted ideas of how people actually perform their work activities.

The approach produces a framework called *worklets* to provide an extensible repertoire of self-contained selection and exception-handling processes, coupled with an extensible *ripple-down* rule set. Using a Service-Oriented Architecture (SOA), a selection service provides workflow flexibility and adaptation by allowing the substitution of a task at runtime with a sub-process, dynamically selected from its repertoire depending on the context of the particular work instance. Additionally, an exception-handling service uses the same repertoire and rule set

framework to provide targeted and multi-functional exception-handling processes, which may be dynamically invoked at the task, case or specification level, depending on the context of the work instance and the type of exception that has occurred. Seven different types of exception can be handled by the service. Both expected and unexpected exceptions are catered for in real time.

The work is formalised through a series of Coloured Petri Nets and validated using two exemplary studies: one involving a structured business environment and the other a more creative setting. It has been deployed as a discrete service for the well-known, open-source workflow environment YAWL, and, having a service orientation, its applicability is in no way limited to that environment, but may be regarded as a case study in service-oriented computing whereby dynamic flexibility and exception handling for workflows, orthogonal to the underlying workflow language, is provided. Also, being open-source, it is freely available for use and extension.

Contents

Keywords ii

Abstract iii

Contents v

List of Figures x

Statement of Original Authorship xiv

Acknowledgements xv

1 Introduction 1

1.1 Problem Area 2

1.2 Problem Statement 7

1.2.1 Limitations of the Paradigm 7

1.2.2 Lack of Support for Exception Handling 8

1.2.3 Lack of System-based, Intelligent Recovery Processes 9

1.2.4 Difficulty in Process Expression 10

1.3 Solution Criteria 11

1.3.1 Support for Flexibility 11

1.3.2 Process Focus for Exception Handling 12

1.3.3 Comprehensibility 13

1.3.4 Locality of Change 14

1.3.5 Support for Verification . . . 14
1.3.6 Support for Reusability . . . 14
1.3.7 Support for Reflection . . . 15
1.3.8 Support for Design . . . 15
1.3.9 Reflection of Actual Work Processes . . . 16
1.4 Research Objectives and Approach . . . 17
1.5 Publications . . . 19
1.6 Original Contributions . . . 20
1.7 Outline of Thesis . . . 21

2 Workflow Flexibility & Exception Handling Issues 23
2.1 Current State of the Art . . . 23
2.2 Conceptual Frameworks vs. Actual Business Processes . . . 29
2.3 Modelling Dynamic Workflows . . . 32
2.4 Exception Handling Techniques . . . 34
2.5 Rules-Based Approaches . . . 36
2.6 Automatic Evolution of Workflow Specifications . . . 38
2.7 Case Handling . . . 40

3 Theoretical Framework: Activity Theory 44
3.1 Issues in Representing Work Practices Procedurally . . . 44
3.2 An Introduction to Activity Theory . . . 46
3.2.1 Background . . . 46
3.2.2 How Activity Theory Describes Work Practices . . . 47
3.3 Applicability of Activity Theory to Workflow Systems . . . 52
3.4 Principles Derived from Activity Theory . . . 53
3.5 Functionality Criteria Based on Derived Principles . . . 55
3.5.1 Criterion 1: Flexibility and Re-use . . . 55

3.5.2 Criterion 2: Adaptation via Reflection 57
3.5.3 Criterion 3: Dynamic Evolution of Tasks 59
3.5.4 Criterion 4: Locality of Change 60
3.5.5 Criterion 5: Comprehensibility of Process Models 60
3.5.6 Criterion 6: The Elevation of Exceptions to “First-Class Citizens” . . . 61
3.6 Functionality Criteria vs. Solution Criteria 63

4 Conceptualisation of Worklets 65

4.1 Worklets — An Introduction 66
4.2 Context, Rules and Worklet Selection 68
4.3 The Selection Process 73
4.4 Exception Handling 76
4.4.1 Exception Types 77
4.4.2 Exception Handling Primitives 79
4.5 Service Interface 82
4.5.1 Selection Interface 82
4.5.2 Exception Interface 85
4.6 Secondary Data Sources 87

5 Formalisation 91

5.1 Flexibility: The Selection Service 92
5.2 Exception Handling: The Exception Service 97
5.3 Evaluating the Rule Tree 100
5.4 Executing an Exception Handler 102
5.5 Suspending a Case 105
5.6 Continuing a Case 106
5.7 Removing all Cases 107
5.8 Compensation 108

6 Implementation **111**

6.1 Service Overview . . . 111

6.1.1 The Selection Service . . . 112

6.1.2 The Exception Service . . . 114

6.2 Service Oriented Approach . . . 117

6.3 Worklet Service Architecture . . . 120

6.3.1 The *WorkletService* class . . . 123

6.3.2 The *selection* Package . . . 123

6.3.3 The *exception* Package . . . 125

6.3.4 The *rdr* Package . . . 127

6.3.5 The *admin* Package . . . 130

6.3.6 The *support* Package . . . 131

6.3.7 The *jsp* Package . . . 134

6.4 Service Installation and Configuration . . . 136

6.4.1 Configuration — Service Side . . . 137

6.4.2 Configuration — Engine Side . . . 138

6.5 Worklet Process Definition . . . 139

6.6 Exlet Process Definition . . . 143

6.6.1 Constraints Example . . . 145

6.6.2 External Trigger Example . . . 147

6.6.3 Timeout Example . . . 149

6.7 Ripple Down Rule Sets . . . 152

6.8 Extending the Available Conditionals . . . 154

6.9 The Rules Editor . . . 157

6.9.1 The Toolbar . . . 157

6.9.2 Adding a New Rule . . . 160

6.9.3 Dynamic Replacement of an Executing Worklet . . . 164

6.9.4 Creating a New Rule Set and/or Tree Set 165
6.9.5 Drawing a Conclusion Sequence 169
6.9.6 Rules Editor Internal Structure 171

7 Validation **176**

7.1 Exemplary Study - Business Environment 177
7.1.1 Background 177
7.1.2 The first:utility Approach 178
7.1.3 A Worklet Service Solution 183
7.2 Exemplary Study - Creative Environment 191
7.2.1 Process: Post Production 193
7.2.2 Process: Visual Effects Production 200

8 Conclusion **207**

A CPN Declarations **213**

B Worklet Selection Sequence Diagrams **216**

C Sample Audit Log File Output **219**

C.1 Selection Service Sample 219
C.2 Exception Service Sample 220

Bibliography **221**

List of Figures

1.1 Example of a ‘static’ workflow process model 3

1.2 Mapping of Solution Criteria to Key Problem Areas 17

2.1 A Chapter Overview 24

2.2 Components of a learning system (from [139]) 39

3.1 Vygotsky’s Model 47

3.2 The Activity System 50

3.3 Aspects of Collective Activity 53

3.4 Relating Derived Functionality Criteria to the Solution Criteria 63

4.1 Conceptual Structure of a Ripple Down Rule (*Assess Claim* Example) 70

4.2 Simple Insurance Claim Model 73

4.3 A ‘view’ of Figure 4.2 extrapolated from the modified RDR for task T_2 75

4.4 Required Selection Interface 84

4.5 Required Exception Interface 86

4.6 ORM Diagram of a Typical Engine Process Log 87

4.7 ORM Diagram of a Process Log for the Worklet Service 88

5.1 CPN: The Selection Service 93

5.2 CPN: The Exception Service 98

5.3 CPN: Evaluating the Rule Tree 101

5.4 CPN: Execution of an Exception Handler 103

5.5 CPN: Case Suspension 105
5.6 CPN: Case Continue 107
5.7 CPN: Remove All Cases 108
5.8 CPN: Compensation 109

6.1 YAWL Language Notation 112
6.2 The Welcome Page for the Worklet Service 113
6.3 YAWL External Architecture 114
6.4 Process – Exlet – Worklet Hierarchy 117
6.5 Interface X Architecture 119
6.6 External Architecture of the Worklet Service 120
6.7 Excerpt of a Rules File 121
6.8 Internal Architecture of the Worklet Service 122
6.9 A Java Package View of the Worklet Service 122
6.10 The WorkletService Class 124
6.11 The ExceptionService Class 126
6.12 The Raise Case Level External Exception Page 135
6.13 The Worklet Service Configuration File (detail) 138
6.14 The YAWL Engine Configuration File (detail) 138
6.15 The YAWL Worklist Configuration File (detail) 139
6.16 Parent 'Casualty Treatment' Process 140
6.17 The 'Treat Fever' Worklet Process 141
6.18 The Update Task Decomposition dialog for the Treat task 142
6.19 The Net-level variables for the TreatFever worklet 143
6.20 Example Handler Process in the Rules Editor 144
6.21 The Organise Concert Example Process 145
6.22 Effective Composite Rule for Do Show's Pre-Item Constraint Tree 146
6.23 Work items listed after exlet invocation for Organise Concert 147

6.24 Workflow Specifications Screen, OrganiseConcert case running 148
6.25 Case Viewer Screen for a running OrganiseConcert case (detail) 148
6.26 Available Work Items after External Exception Raised 149
6.27 The *Timeout Test 3* Specification 150
6.28 Rules Editor Showing Single Timeout Rule for *Receive Payment* Task 151
6.29 Rule detail for *Receive Reply* 151
6.30 Available Work After *CancelOrder* Worklet Launched 152
6.31 Example rule tree (ItemPreConstraint tree for DoShow task) 153
6.32 Rules Editor Main Screen 158
6.33 The Toolbar 159
6.34 Rules Editor (Add New Rule Screen) 162
6.35 The Choose Worklet dialog 164
6.36 Dialog Offering to Replace the Running Worklet 165
6.37 Example Dialog Showing a Successful Dynamic Replacement 165
6.38 Create New Rule Set Form 166
6.39 Creating a New Rule Tree 168
6.40 The Draw Conclusion Dialog 170
6.41 A Conclusion Sequence shown as Text (detail) 171
6.42 Rules Editor Classes 172
6.43 The frmDrawConc class 173
6.44 The RuleSetMgr class 174
6.45 The Rule Classes 175

7.1 Process Model to Withdraw a WLR Service 180
7.2 Process Model to Compensate for the Cancellation of a WLR Withdrawal . . . 182
7.3 A Worklet Solution for the WLR Withdrawal Process 183
7.4 ItemPostConstraint Exlet for PlaceOrder task 184
7.5 Compensation Worklet for a failed WLR Withdrawal Order 184

7.6 WithdrawOrderFailedAfterPlace Worklet 185
7.7 Compensatory Worklet PreWithdrawCancel 187
7.8 Compensatory Worklet CancelOrder 187
7.9 Compensatory Worklet CancelOrderWithRollback 188
7.10 CaseExternal rule tree for the WithdrawOrder process 189
7.11 ItemPostConstraint rule tree used by multiple tasks 190
7.12 Selection rule tree for the failedOrder task of the OrderFailed process 190
7.13 Static Post Production Master Process (C-EPC language) 194
7.14 Static Post Production Master Process (YAWL language) 197
7.15 Post Production Master Process (Worklet-Enabled) 197
7.16 Selection Rule Tree for the PrepareForEdit task 198
7.17 Worklets ppPrepareFileForEdit and ppPrepareTapeForEdit 198
7.18 Selection Rule Tree for the Finalise Task 199
7.19 Static Visual Effects Production Process (C-EPC language) 201
7.20 Static Visual Effects Production Process (YAWL language) 202
7.21 Visual Effects Production Parent Process (Worklet Solution) 203
7.22 Visual Effects Worklets 204
7.23 Approve workitem of Animation Worklet in YAWL Worklist (detail) 205
7.24 Rule Sets for VFX Process and Repertoire Worklets 206

B.1 Sequence Diagram for Workitem Substitution 216
B.2 Sequence Diagram for Worklet Replacement 217
B.3 Sequence Diagram for Worklet Completion 217
B.4 Sequence Diagram for Workitem Cancellation 218

Statement of Original Authorship

The work contained in this thesis has not been previously submitted to meet requirements for an award at this or any other higher education institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made.

Signed: ____________________________________

Date: ____________________________________

Acknowledgements

I would like to gratefully acknowledge and thank my principal supervisor, Associate Professor Arthur ter Hofstede, for his unflagging support, guidance and assistance. Similarly, my associate supervisors Dr David Edmond and Professor Wil van der Aalst for their support and feedback.

I would also like to thank Dan Simkins and Andrew Hastie of first:utility for kindly supplying two of their operational workflow specifications to this research to be used as part of a comparative study with the worklet service. Also, thanks must go to Stefan Seidel and Johannes Lux for providing the C-EPC models to this research referred to in the validation chapter.

Of course, I must also acknowledge the support and encouragement of my family, without whom this thesis would not exist.

Chapter 1

Introduction

Organisations are constantly seeking efficiency improvements for their business processes in terms of time and cost. To help achieve those goals, many are turning to Workflow Management Systems (WfMSs) to configure and control those processes [2, 62, 75]. WfMSs are software systems supporting the modelling, analysis, enactment and management of business processes [126, 47]. The key benefits organisations seek by implementing Business Process Management and Workflow solutions include [93]:

- Improved efficiency: automation of business processes often results in the elimination of many unnecessary steps;
- Better process control: improved management of business processes achieved through standardising working methods and the availability of audit trails;
- Improved customer service: consistency in processes leads to greater predictability in levels of response to customers;
- Flexibility: software control over processes enables their re-design in line with changing business needs; and
- Business process improvement: focus on business processes leads to their streamlining and simplification.

The use of WfMSs has grown by concentrating on modelling rigidly structured business processes that in turn derive well-defined workflow instances [34, 98, 143]. However, the proprietary process definition frameworks imposed make it difficult to support (i) dynamic evolution

and adapatation (i.e. modifying process definitions during execution) following unexpected or developmental change in the business processes being modelled [40]; and (ii) deviations from the prescribed process model at runtime [148, 47, 70].

But change is a 'given' in the modern workplace. To remain effective and competitive, organisations must continually adapt their business processes to manage the rapid changes demanded by the dynamic nature of the marketplace or service environment. Also, a large proportion of workplaces undertake activities that do not easily conform to rigid or constricting representations of their work practices. And even in the most concrete processes deviations will occur within almost every instantiation.

If WfMSs could be extended to meet the challenges of evolutionary and unexpected change in business processes, then their applicability would widen to include a far greater proportion of workplaces. Such support would not only benefit existing users of workflow technologies, but would also introduce those businesses which employ more creative or ad-hoc processes to the range of benefits that workflow systems offer.

This research offers a solution which meets that challenge.

1.1 Problem Area

A workflow can be defined as a composite set of tasks that comprise coordinated computer-based and human activities [118, 114]. A workflow model or schema is a formal representation of work procedures that controls the sequence of performed tasks and the allocation of resources to them [134, 25].

The development of a workflow model typically begins with an analysis of current procedures and processes. Subsequently, a model is developed based on those practices and business rules, then input into the WfMS and repetitively executed, supporting and giving formal structure and flow control to those processes. An audit history of each execution may be stored and analysed as a basis for seeking further improvements and efficiencies. Such improvements invariably involve remodelling the work process to incorporate the changes.

However, translating abstract concepts and descriptions of business practices and rules into process models is a far from trivial exercise, but involves a creative process which includes progressive cycles of refinement between management, workers and technical experts [62]. There

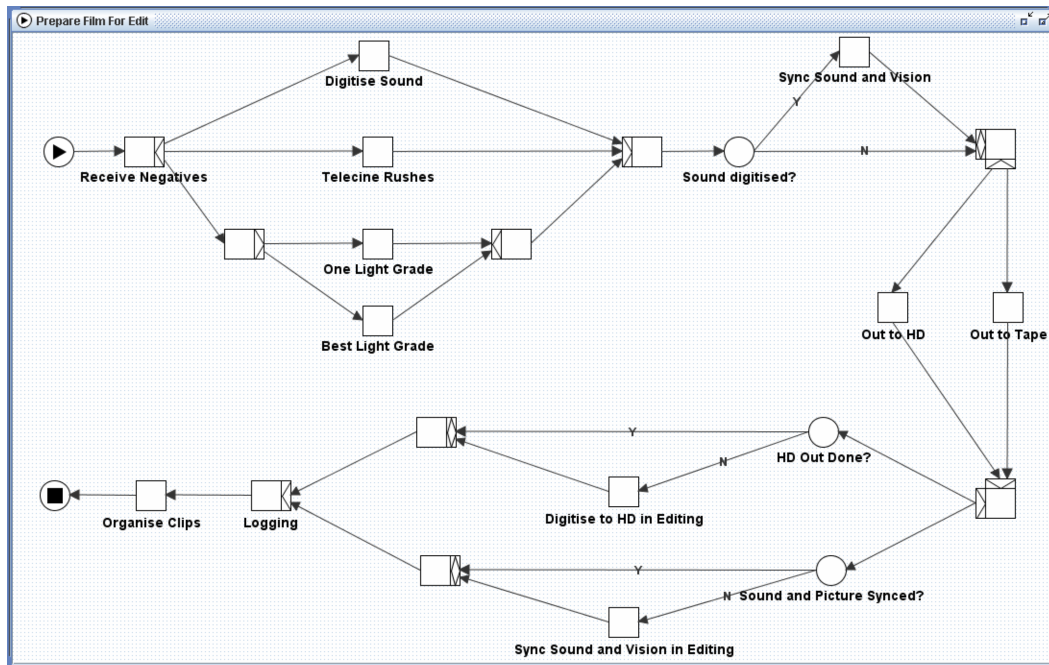

Figure 1.1: Example of a 'static' workflow process model. Even though the work process being modelled here is not especially complex, there are many points in the process where decisions or choices must be made on which branch of the model to execute, based on the conditions of the particular executing instance. That is, any flexibility must be incorporated into the process control flow as explicit conditional branches, so that control flow decisions are mixed with business process logic

are sizeable development costs involved in mapping an abstract process to a logical schema, which must be weighed against the perceived cost benefits that the workflow system will deliver. Therefore, current workflow systems are most advantageous where they provide support for standardised, repetitive activities that do not vary between execution instances.

In some work environments, and for some kinds of activities, formal representations of work play a fundamental role by giving order to tasks and assisting in getting the work completed correctly (for example medical and banking environments). But even in such highly structured environments, it is difficult (if not impossible) to successfully capture all work activities, and in particular all of the task sequences possible, in a workflow model, at least not without producing some very complex models.

In addition to the complexities involved in representing a work process in a computationally executable way, it is the case that, for any given human activity, the process for successfully completing the activity is constantly evolving. A worker, or group of workers, given responsibility for a particular task, will naturally seek to minimise the amount of work necessary for completing the task by deriving new, more efficient methods. Also, change can be introduced

via many other sources, including government regulation, new competitors, new markets, improvements in plant and equipment, workforce and resource availability and so on.

Where a work process is mediated by a workflow model, that model represents at best a point in time or snapshot of the particular process being modelled. Thus, there is generally no flexibility allowed in the system to accommodate the natural evolution of the work process, nor the evolution of organisational goals. Manual interventions into the workflow process become increasingly frequent as staff attempt to manipulate workflow inputs and outputs to conform with changes in workplace practices. These manual intrusions necessitate reduced productivity and increased processing time, and can lead to farcical situations such as those quoted by Strong and Miller [160]:

- Staff routinely correcting computer-generated inventory data because it was too inaccurate for decision making;
- Staff entering incorrect parameters into a system so it would return 'correct' results; and
- Staff routinely correcting inappropriate, computer-generated plant assignments before forwarding the data on to the correct plant.

Since it is undertaken in an ad-hoc manner, manual handling incurs an added penalty: the corrective actions undertaken are not added to 'organisational memory' [112, 17], and so natural process evolution is not incorporated into future iterations of the process.

In other work environments, such as those where activities are more creatively focussed, formal representations may provide merely a contingency around which ad hoc tasks can be formulated [25]. Barthelmess et al. state that "In real life, both in office and scientific lab environments, the enactment of any workcase may deviate significantly from what was planned/-modelled" [30]. Thus, adherence to formal representations of task sequences ignores, and may even damage, the informal work practices that also occur in any set of activities [25].

In addition, WfMSs generally assume a two-phase approach, where the modelling phase must be fully concluded before any execution instances can be started [30]. But this approach ignores those organisational environments where many of the necessary tasks needed to complete an activity cannot be known beforehand. Since only the most standardised processes can be fully modelled with the current WfMSs, a vast number of organisations are unable to take advantage of the possible benefits workflows could bring.

To summarise, within the workflow field a major source of problems has been the prescriptive nature of workflow specifications, which do not allow for unanticipated variations [40]. Current workflow systems provide tools for modelling business processes that are predictable and repetitive. But in many environments, business processes vary highly and are dynamic with regards to their actual execution [143], and may differ greatly between individual instances. Therefore, for a WfMS to be most effective, it must support dynamic change (i.e. during execution).

In fact, it is because of the discrepancies between real-world activities and formal representations of them that workflow process instances typically experience deviations or exceptions during their execution. Traditionally, exceptions are understood to be events that by definition occur rarely. But in a workflow process, an exception can be described as an event that is deemed to be outside 'normal' behaviour for that process. Rather than being an error, it is simply an event that is considered to be a deviation from the expected control flow or was unaccounted for in the original process model. Such events happen frequently in real working environments [90].

Exceptions are a fundamental part of most organisational processes [104]; in fact, a substantial proportion of the everyday tasks carried out in a business can be categorised as exception handling work [30]. The distinction between what can be considered normal operational behaviour and what is an exception is not absolute, but separating out those activities that do not occur very often helps to focus on the normal or central flow of control [114]. This perspective implies that it is the frequency of the occurrence of an event that delineates a normal event from an exceptional one, rather than there being any suggestion of error involved.

From a humanistic perspective, every executing instance of a work process will incorporate some deviation from the plan. In this sense, a work plan can be seen to be just another resource or tool which mediates the activities of workers towards their objective, rather than a prescriptive blueprint that must be strictly adhered to. Such deviations from the plan should not be considered as errors, but as a natural and valuable part of the work activity, which provides the opportunity for learning and thus evolving the plan for future instantiations.

Historically, exception handling within WfMSs has fallen well short, particularly after execution has commenced [104]. Current WfMSs provide little, if any, support for exception management, and are typically unable to effectively manage the types of exceptions that are peculiar to workflows [47].

Current thinking is that if an exception is expected, or could conceivably have been anticipated, then the modeller has *a priori* knowledge of such an event, and it should therefore be built into the model. However, if a modeller builds all possible *a priori* exception scenarios into a model, it can lead to some very complex models, much of which will never be executed in most cases. Such approaches add orders-of-magnitude complexities to the original workflow logic, as well as making the original process barely recognisable, if at all. Mixing business logic with exception handling routines complicates the verification and modification of both [81], in addition to rendering the model almost unintelligible to most stakeholders.

Conversely, if the exception is unexpected, or could not possibly have been anticipated given the scope of knowledge of the work process available at the time the model was designed, then the modeller has, by definition, no prior knowledge of the event, and therefore the model is deemed to be simply deficient, and thus needs to be amended to include this previously unimagined event (see for example [50]). This view, however, tends to gloss over or ignores the frequency of such events, or the costs involved with their correction. Such exceptions occur more frequently in processes that are very complex and/or with a high variability. When they occur, they are normally captured and managed by human agents, typically by halting process execution [49], which naturally has a negative impact on business efficiency and competitiveness.

So it can be seen that current methods for exception handling in WfMSs contravene accepted programming paradigms of modularity, encapsulation and reusability. Most modern programming languages provide exception handling mechanisms that separate the exception handling routine from the 'normal' program logic, which facilitates the design of readable, comprehensible programs [81]. 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 workflow model. Any unexpected exceptions that occur either require the intervention of a human agent, or result in a process abort. Since most 'real-world' workflow processes are long and complex, neither human intervention nor terminating the process are satisfactory solutions [81].

Thus a large group of business processes do not easily map to the rigid modelling structures provided [24], due to the lack of flexibility inherent in a framework that, by definition, imposes rigidity. This inflexibility extends to the management of exceptions, which places further limits on how accurately a workflow model can reflect the actual business process it is based on.

Rather, process models are 'system-centric', meaning that work processes are *straight-jacketed* [16] into the paradigm supplied, rather than the paradigm reflecting the way work is actually performed [34]. As a result, users are forced to work outside of the system, and/or constantly revise the process model, in order to successfully complete their activities, thereby negating the perceived efficiency gains sought by implementing a workflow solution in the first place.

1.2 Problem Statement

This research has identified four key problem areas, each described in the following subsections, that limit the applicability and reach of workflow technologies. The divisions between them are not distinct — all are related to the rigidity enforced by the inflexible modelling frameworks employed by WfMSs, and the consequent difficulties in placing more dynamic, information intensive processes within that framework.

1.2.1 Limitations of the Paradigm

Workflow Management Systems provide vast potential in supporting business efficiencies and workplace change. However, as discussed, these efficiencies have thus far been limited to work environments that have relatively rigid work processes — that is, processes that do not vary between execution instances.

Because of this enforced rigidity, workflow management systems do not handle well:

- exceptional events;
- developmental or unexpected change;
- creative or ad-hoc processes.

These limitations would be relatively inconsequential if all work processes were reasonably static. However, such rigid work processes are limited to a small subset of actual work environments. Most business practices either evolve extemporaneously, need to react swiftly to market changes, or simply employ practices that have a degree of flexibility inherent in them. As a result, a large number of business environments are currently limited in the degree to which

workflow solutions can be successfully employed within their organisations, and therefore are unable to enjoy the benefits and business efficiencies that a workflow solution may bring.

These limitations may be attributed in part to the manufacturing paradigm employed by traditional WfMSs, and the fact that there are many aspects where administrative and service processes differ from manufacturing processes [7] (also see Section 3.1).

While some workplaces have strict operating procedures because of the work they do (for example, air traffic control) many workplaces have few pre-specified routines, but successfully complete activities by developing a set of informal tasks that can be flexibly and dynamically combined to solve a large range of problems [17].

Human work is complex and is governed by rules often to a much lesser extent than computerised processing. Because of this, techniques applied to modelling work processes must avoid making assumptions about regularities in work procedures [103]. Most current workflow systems use their own conceptual framework, which is usually postulated on programming constructs rather than theoretically founded [100]. Therefore, a new paradigm is needed, based on a sound theoretical foundation, that moves away from the rigidity inherent in current workflow management systems, and towards a system that incorporates inevitable change as a fundamental building block in the development of business process and workflow models, and one that better reflects the way work activities are actually carried out.

1.2.2 Lack of Support for Exception Handling

The interpretation, particularly held by managers [7], that exceptions are uncommon is not supported by research. In fact, exceptions are a common and fundamental component of any business process [160]. However, because of the proprietary frameworks involved, exceptions are largely uncatered for in workflow models, and are therefore most often handled off-system, typically by halting the process until the exception is dealt with by a worker.

As a result, exception handling is typically carried out completely outside the structure of the workflow that was originally designed to make a business process execute more efficiently. As exceptions occur in almost every execution of an activity, and represent a learning opportunity, it is simply unacceptable for exceptions to be dealt with off-system and therefore go unrecorded.

The traditional approach towards exception handling has either been to consider them as errors, and therefore abnormal events that the system should not have to deal with, or to build

them directly into a monolithic process model which quickly becomes almost indecipherable and unable to be maintained. In this approach, exceptions are always handled at the macro level; there is no concept of distributed exception handling, and therefore easier maintenance, within the micro levels of a process.

It would therefore be useful to provide an approach that is seen as a tool to assist organisations dealing with exceptional events as a normal and welcomed part of every activity instance, rather than the inverse view of exceptions being a hindrance to the normal execution of a workflow. Instead of attempting to predict, and thus impose methods to handle all possible exceptions before a workflow is implemented, a better approach would be found in a system where the inevitability of exceptional behaviours is an accepted given, and develop a set of procedures and activities that can deal with them in a robust manner [169] *as they occur*.

A workflow system that provides such support for exceptions would thereby necessarily provide the ability to better reflect actual work practices and processes, and would therefore have applicability to a wider variety of organisational environments.

1.2.3 Lack of System-based, Intelligent Recovery Processes

Typically, whenever a workflow process model is designed, there is a trade-off on the spectrum between including (i) every conceivable event that may occur into the model, no matter how unlikely, thereby increasing complexity and decreasing readability of the model, and including (ii) only those events that are considered 'normal' events for the process, thereby choosing to ignore, or not provide support for, any deviations that may occur. Even if the former approach is taken, the model designer is limited to include only those exceptions for which *a priori* knowledge exists.

When an exception occurs, and it is then handled off-system, there exists no system record of how the exception was handled. If the exception could be handled on-system, then not only would there be no need to halt the process while the exception was handled, but a record would be kept on the system so that the same (or similar) handling method could be employed in the event of the same or similar exception reoccurring in the future — thus the handling method would dynamically become an implicit part of the model for all future instantiations of it. If the particular exception handler could be selected by the system itself, based on rules and/or archived execution histories and the context of the process instance, then further efficiencies

could be gained by removing the need to refer the exception to a human agent.

Since the occurrence of an exception represents a diversion from the 'normal' course of events, and therefore a learning opportunity, a system that records all diversions, and then has the ability to recall the methods employed in dealing with them in future instantiations, would necessarily provide efficient support for flexible work practices.

1.2.4 Difficulty in Process Expression

There is evidence that the nature of workflow systems are typically poorly understood by stakeholders (including users, managers and system analysts), which leads to confusion during the design and implementation of automated workflow processes in an organisation [100].

Typical workflow management systems demand that processes be defined at best partly graphically (i.e. a graphical process editor) and partly textually (code blocks, scripts and so on) — especially in the expression of what to do when an exception occurs. Many systems also demand a process model that is monolithic, so that every possible action is constructed on the same level. This makes for very complex models that are both difficult to construct, and difficult to understand, by *all* stakeholders [44].

A recent survey conducted by the Workflow Management Coalition found that, when asked the single greatest perceived benefit to be realised by a BPM solution, the largest response was for the "ability to visualize, simulate and trouble-shoot business processes before committing [to deployment]" [137]. It is clear that organisations engaging workflow solutions seek systems that offer process modelling support for these activities.

Better acceptance of workflow support systems would be facilitated if the process models were easier to both design and understand. A workflow system that could construct and present all process models graphically (especially exception handling methods) would be of great benefit to system designers, since all of the activity's components would be readily accessible without having to delve into text based languages and scripts that require specialist knowledge. Additionally, a fully graphical model would be more easily understood by those stakeholders without specialist system knowledge. A design that could be more easily created, and better understood by all stakeholders, would increase the levels of efficiency and acceptance of a workflow management solution.

1.3 Solution Criteria

An effective solution to the problems identified would satisfactorily address the following criteria. Note that each of the criteria discussed is considered orthogonal, except where explicitly specified.

1.3.1 Support for Flexibility

The Workflow Management Coalition survey mentioned above also found that 75 per cent of respondents reported they were currently performing work on improving existing processes (up to 92 per cent for the Finance sector) and 56 per cent were currently involved in a major business process redesign [137]. Such statistics underscore the frequency of organisational change and importance of providing a workflow system which supports flexibility and the ability to adapt to change.

To provide maximum flexibility, the solution should support the ability to efficiently add, remove, cancel and/or modify tasks, sequences and so on. One way to achieve this would be to modularise the work process being modelled into a 'parent' or manager level model and a series of suitable 'sub-processes'. By using a modular approach, a resultant model may range from a simple skeleton or base to which sub-processes can be added at runtime (supporting dynamic change) or may be a fully developed model representing the complete work process, depending on user and organisational needs.

Depending on the particular circumstances, modifications to workflow process models may need to be applied once only (i.e. during the execution of a specific instance), or to all currently running instances and/or any future instances, or a change may need to be made to the process model itself. To support the idea of modularity, the solution would provide for sub-processes to be added to the current running instance only, one or more running instances and/or to the conceptual model for future executed instances.

As far as possible, appropriate modifications to process instances should be invoked with little or no human involvement (i.e. no manual handling) via an extensible rule set. Where that is not possible (for example, an unexpected exception has occurred), a guided choice should be made available to the designer, allowing for the implementation of an appropriate sub-process chosen from a catalog of available processes, ensuring the framework matches as closely as

possible the automated substitution method. A designer may be guided to choose certain sub-processes over others based on system 'intelligence' using contextual instance data, archival records and heuristic algorithms. Support should also be provided to the designer to allow a new sub-process to be specified and added to the model 'on-the-fly'. Such an approach will minimise the likelihood of tasks needing to be performed off-system.

1.3.2 Process Focus for Exception Handling

One of the limiting factors of previously expounded solutions to the problem of the representation of workflow exceptions at the conceptual level, and the detection, capturing and handling of them during execution, is the view that exceptions are to be considered as errors or problems. As such, they are seen to be annoyances which either should have been foreseen and therefore prevented from occurring in the first place, or perceived as impossible to predict and therefore best left handled off-system.

An effective solution to the problem of workflow exception representation and execution will regard exceptions not as errors, but as a normal and valuable part of every work activity, which can be used to refine or evolve the process towards improved efficiency. As a result, exceptions should be implemented in a positive fashion to better reflect (within the workflow model) the actuality of the work process it supports.

Therefore, the modelling methodology developed should have an exception-centric perspective. Rather than exception occurrences being considered errors, or outside what is considered to be 'normal' behaviour, exceptions should be considered as valuable parts of the process. Therefore, models should reflect the naturality of exceptions within any given process, and models should be able to be developed, from the ground up, with exceptions at the centre.

Since there is a class of exceptions that cannot be known when the execution of a workflow process instance begins, an effective solution must provide the ability to quickly source an appropriate exception handler, then incorporate it into the running instance. That is, exception handling techniques should be available at the conceptual and execution levels, thereby supporting the accepted programming paradigms of modularity, encapsulation and reusability.

1.3.3 Comprehensibility

A major limiting factor in the uptake of workflow solutions is the complexity of the models developed for all but the most trivial work activities. Workflow Management Systems that support or require the development of complex models fail to take into account that:

- the more complex the model, the more difficult it is to modify, enhance or evolve it;
- understandability suffers when a model contains many possible paths represented on a single plane; and
- there may be many stakeholders with a vested interest in the representation of a work activity by a workflow model, all with differing perspectives and levels of comprehensibility.

The question of the relative involvement of the various stakeholders in the development of the process is an important one. While the modeller will have a detailed understanding of the components and structure of the model, a manager will generally have a more holistic view and thus will not have the expertise to decipher it, nor will a staff member charged with carrying out a particular task. However, the acceptance of the workflow solution may hinge on being able to provide management with a modelled representation of a work process which is comprehensible to them [45].

So, a solution to the problem will provide a modelling methodology that can be easily understood by **all** stakeholders. Therefore, complexity at the conceptual design phase should be minimised.

To aid comprehensibility, exception processes should be able to be represented and constructed graphically to the greatest extent possible, as opposed to textual representation, which is typically the only option available in commercial WfMSs for exception representation. As such, the solution should support the idea of *graphical conceptuality* — the designer should not be forced to write code or scripts to achieve a desired outcome. Additionally, exception handling processes should be formulated and stored externally to the process model they serve — that is, to separate the exception handling processes from the parent workflow.

1.3.4 Locality of Change

Related to support for flexibility is support for *locality of change*. Bass et al. [31] suggest that increasing the adaptability of software is largely a function of how localised necessary modifications can be made. In terms of workflow process models, this idea suggests two desirable and related goals. Firstly, to ensure that a workflow process model is strongly adaptable, modifications should be able to be limited to as small a number of components as possible. Secondly, any changes made within those components should impact on other components to the least extent possible.

A solution to this criterion can achieve adaptability by strongly providing for locality of change. One possible approach is, as previously discussed, providing for the definition of a model as a set of sub-processes, each of which is a distinct entity. By providing support for such modelling structures, locality of change will be achieved by allowing a sub-process to be added, removed or modified without the need to change the entire model. Also, since each sub-process would be fully encapsulated, any changes made to a sub-process should not impact other sub-processes belonging to the model.

By taking a modular approach, each portion of the model would be verifiable for correctness in its own right, independently to the overall process.

1.3.5 Support for Verification

Leading on from the previous criterion, a model should be easily verifiable for correctness. The modelling paradigm employed should have a formal theoretical grounding and there should be some form of system support for automated verification. However, any verification methods developed will need to be balanced against ease of use considerations.

1.3.6 Support for Reusability

Exception handling and substitution sub-processes should be defined using the same modelling constructs as a 'normal' process. That is, as far as the modelling system is concerned, a sub-process representing a standard event and one representing an exceptional event should be able to be constructed identically.

If sub-processes can be defined in this manner, it will provide for reusability of processes,

since a sub-process which forms part of a larger, central process, may be called more than once within that central process, or may be used by several distinct processes, without any modification required to the sub-process in question. Also, a sub-process may be used as part of the 'normal' flow in one instance, and as an exception handler in another (and vice versa).

In addition, by taking the modular approach, the sub-processes can be viewed as 'building blocks' which will provide a template from which new sub-processes can be created with minimal effort by the designer.

1.3.7 Support for Reflection

The ability of the problem solution to 'suggest' possible handlers for certain exceptional events as they occur is vital. Exceptional events should be handled using a two phase approach: if, based on a set of conditions, contexts, archival data and so on, a matching handling process can be found for a particular event, then that process should be automatically invoked. If no such match can be found, then a list of possible processes, selected from a catalog of processes by applying heuristic techniques, should be presented to a modeller from which an appropriate choice can be made.

Similarly, a set of sub-processes could be made available for the purposes of providing flexibility. Depending on the current context of the workflow instance and available external data sources, a choice can be made to execute the most appropriate sub-process for any particular situation.

The construction of knowledge of workflow events using discovery methods based on archival data and context will form an effective tool in the suggestion phase of the problem solution.

1.3.8 Support for Design

The solution should provide support at the process design phase to assist the designer to determine what events should be modelled in the main process, and what are to be modelled as exceptional events. In essence, there is no difference between a process that defines 'normal' events, and a process that models exception handling procedures. Therefore, the designer should be free to pre-define a set of processes within the process definition, and/or leave the specification of particular sub-process usages to the execution phase. Further, the degree to

which individual process models are pre- or post-defined should be flexible, to be implemented by the designer after consideration of the relevant circumstances of the work process and the demands of the organisational environment.

1.3.9 Reflection of Actual Work Processes

An important consideration that the solution will need to address is whether it sufficiently synchronises with the ways in which work processes are actually planned and realised. That is, to the greatest extent possible, the solution should be able to reflect actual human work processes, rather than enforce representations of how work is carried out to meet the limitations of the manufacturing metaphor, the modelling language, or the skills of the designer.

Such considerations will need to take into account that while work processes are rigid and resistant to change in some work environments, in others processes are fluid and highly changeable, and in fact encouraged to be so.

Figure 1.2 shows the relationship between the nine solution criteria discussed in this section and the four key problem areas described in Section 1.2. It is apparent that several of the criteria span more than one problem area, reflecting the fact that the divisions between the four problem areas are not distinct, while other criteria are directly related to a particular problem area. However, each criterion should be regarded as equally important to an effective, overall solution.

For example, a solution that provided support for flexibility (such as the ability to add/remove/modify tasks and sequences) would successfully address the 'Limitations of the Paradigm' area, while a solution that regarded exceptions as a normal and valuable part of work activities would solve the problem of 'Lack of Support for Exception Handling', as detailed in the earlier sections of this chapter. Further, a solution that was built on a framework that reflected actual work processes would address all four problem areas, since it would need to break free of the rigid paradigms imposed, provide process expressiveness to successfully mirror actual work practices, and allow the system to record, reflect on and recover from deviations in process instances.

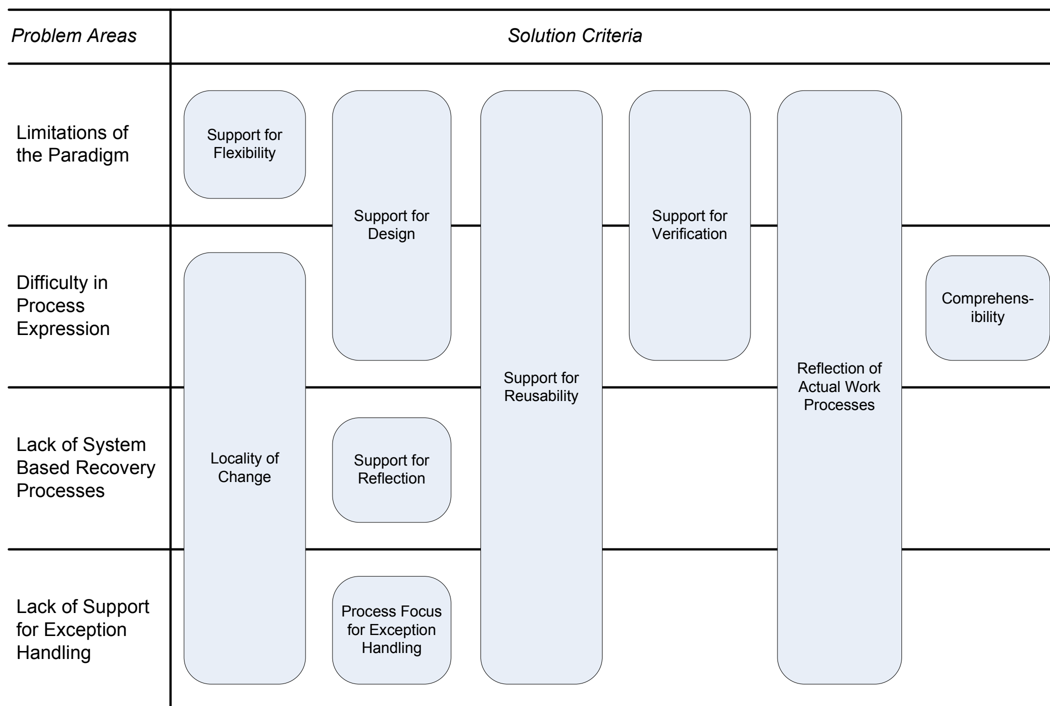

Figure 1.2: Mapping of Solution Criteria to Key Problem Areas

1.4 Research Objectives and Approach

The aim of this research is to conceptualise, formalise, implement and validate a solution to the issues outlined in Section 1.2 and, in doing so, satisfy the solution criteria discussed in the previous section.

In order to seek a viable solution to the related issues of workflow flexibility and exception handling discussed, this thesis begins by surveying related work to situate the scope of the research. It then examines the fundamental theoretical contexts underpinning work practices and derives a set of principles that summarise organisational work activities. Establishing those basic principles provides the foundations for a new direction that moves away from the production-line paradigm and therefore better mirrors all work environments, so that the applicability of workflow technologies can be widened.

In developing a solution to the problem, this research provides:

- new and modified methods with a sound theoretical foundation at the conceptual level;
- the integration of those methods into a conceptual solution that addresses both flexibility and exception handling using the same constructs;
- a formalisation of the approach using Coloured Petri Nets;

- the realisation and deployment of a discrete *Worklet* service that demonstrates the solution; and
- exemplary studies of two disparate work environments that serve to validate the approach.

The primary deliverable of this research is the *Worklet Service*, a discrete web-service that has be built using a Service-Oriented Architecture, so that it can be implemented orthogonal to the underlying workflow enactment engine. The Worklet Service transforms otherwise static workflow processes into fully flexible and dynamically extensible process instances that are also supported by a full range of dynamic exception handling capabilities.

The research identified four major topic areas (as discussed in Section 1.2) which were investigated to determine how workflows are developed and whether seeking a different perspective will produce a more appropriate approach. The approach of this research to those topic areas are:

1. **Limitations of the Paradigm:** Fundamentally, a workflow management system should provide support for the way work is actually carried out (that is, to mirror it as closely as possible), and should use a representational metaphor that allows work practices to be accurately modelled, successfully reflecting the process it represents.

   The way that computers and the processing they do have been conceptualised and internalised by developers and users has led to a stifling of expression in representational methodologies. The production-line or manufacturing paradigm employed by most current WfMSs forces work patterns and processes to 'fit' into often inappropriate representational frameworks. A theoretical understanding of how people actually carry out work and how work plans are realised and adhered to has led to an approach which better supports disparate work environments.

2. **Exception Handling:** The way exceptions are represented and understood may have limited the development of methods to deal with events that are not, for one reason or another, included in the original workflow process. So, new methodologies have been developed to better incorporate exceptional behaviours in workflow specifications. This research examines what exceptions actually consist of, why they occur, and how they can be handled dynamically, with minimal human input.

3. **System-based Recovery Processes:** As discussed previously, WfMSs typically deal with exceptions by halting the process and/or dealing with the situation off-system. However, the service resulting from this research provides for all exceptions, whether expected or not, to be handled online and allows the process to continue in most cases without interruption.

4. **Process Expression:** By developing solutions for the three topic areas mentioned above, a natural secondary effect will include new ways to conceptualise and model workflow processes that are easily validated, understood and maintained.

1.5 Publications

The following publications have been produced while researching this thesis:

- Michael Adams, Arthur H.M. ter Hofstede, David Edmond, and W.M.P. van der Aalst. Dynamic and Extensible Exception Handling for Workflows: A Service-Oriented Implementation. *BPM Center Report BPM-07-03*, 2007. BPMcenter.org.

- Stefan Seidel, Michael Adams, Arthur ter Hofstede and Michael Rosemann. Modelling and Supporting Processes in Creative Environments. In *Proceedings of the 15th European Conference on Information Systems (ECIS 2007)*, St. Gallen, Switzerland, June 2007. *(to appear)*.

- 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.

- Michael Adams, Arthur H.M. ter Hofstede, David Edmond, and W.M.P. van der Aalst. Implementing Dynamic Flexibility in Workflows using Worklets. *BPM Center Report BPM-06-06*, 2006. BPMcenter.org.

- 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 Falcã e Cunha, editors, *Proceedings of the CAiSE'05 Forum*, pages 45-50, Porto, Portugal, June 2005. FEUP Edicoes.

- 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. *BPM Center Report BPM-05-04*, 2005. BPMcenter.org.

- 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.

1.6 Original Contributions

The original contributions of this research to the field are:

- An analysis of existing work within the topic-spaces of flexibility and exception handling for workflows, which led to the definition of nine conceptual criteria that would be satisfied by an effective solution to the issues identified;

- An examination of the historical foundation of the representational metaphor used in workflow systems, which is shown to be a primary factor in the limitations imposed on expressiveness by conceptual frameworks;

- The derivation from Activity Theory of ten fundamental principles that describe the basic tenets of organised work practices with particular relevance to the support of those practices by workflow systems;

- The definition of six functionality criteria that a workflow system grounded in the derived principles would need to satisfy to successfully support actual work practices in a wide variety of work environments;

- A cross-mapping of the six functionality criteria based on derived principles with the nine conceptual criteria relating to issues identified, which serves as a validation of both the derived principles and the original conceptual criteria identified;

- The conceptualisation and formalisation of a workflow service based on the derived principles of work practice;
- The implementation of a novel workflow service that facilitates dynamic flexibility and exception handling of workflows, using a service-oriented architecture, that satisfies the functionality criteria based on the derived principles;
- The application of 'ripple-down' rules to the provision of a contextual knowledge base for the support of dynamic workflow selection and enactment;
- An implementation of dynamic, context-aware exception handling processes;
- A new graphically-based tool that allows the creation and maintenance of ripple-down rule sets, and the design and specification of exception handling processes; and
- Two exemplary studies that serve to validate the approach.

The primary deliverable of this research is the *Worklet Service*, a discrete web-service that has been implemented as a Custom Service for the YAWL workflow environment [12]. However, because it has been designed with a minimal interface 'footprint' and utilising a Service Oriented Architecture, the worklet service is in no way limited to the YAWL environment. The service provides dynamic flexibility and exception handling capabilities and management for the underlying abstract workflow enactment engine. The worklet service is open source and has been released as a member of the YAWL package, downloadable from `www.yawl-system.com`.

1.7 Outline of Thesis

This thesis is organised as follows:

- Chapter 2 positions this research by reviewing the literature on workflow flexibility and exception handling issues, and surveys several major commercial workflow management systems and academic prototypes in relation to those issues.
- Chapter 3 discusses the problems arising from the procedural representation of work practices, which leads into the introduction of *Activity Theory* a set of descriptive principles

of human activity. It defines a set of principles derived from Activity Theory that describe work practices, then maps them to the set of solution criteria discussed in Section 1.3.

- Chapter 4 introduces the conceptualisation of the design of a service which provide solutions to the issues raised, called the *Worklet Service*.
- Chapter 5 presents a formalisation of the service using Coloured Petri Nets.
- Chapter 6 describes the architecture, operation, realisation and deployment of the Worklet Service.
- Chapter 7 validates the design of the resultant service by applying the solution to the workflows of two exemplary studies: one involving a relatively rigid business scenario and the other a more creative environment.
- Chapter 8 concludes the thesis.

Chapter 2

Workflow Flexibility & Exception Handling Issues

This chapter conducts a review of the literature and represents a discussion of the current issues regarding workflow flexibility and exception handling. It also reviews current commercial and academic prototype workflow management systems in relation to their approaches to the identified issues. Figure 2.1 shows an overview of the sections of this chapter and how they inform this research.

2.1 Current State of the Art

This section discusses the levels of support for workflow flexibility and exception handling in several of the leading commercial workflow products and a number of academic prototypes. Unless explicitly stated otherwise, information regarding the products has been gleaned from product manuals, published literature and white papers. The version numbers specified for the commercial products are the versions that were reviewed.

Since the mid-nineties much research has been carried out on issues related to dynamic flexibility and exception handling in workflow management systems. Such research was initiated because, generally, commercial workflow management systems provide only basic support for handling exceptions [152, 126, 82, 52] (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

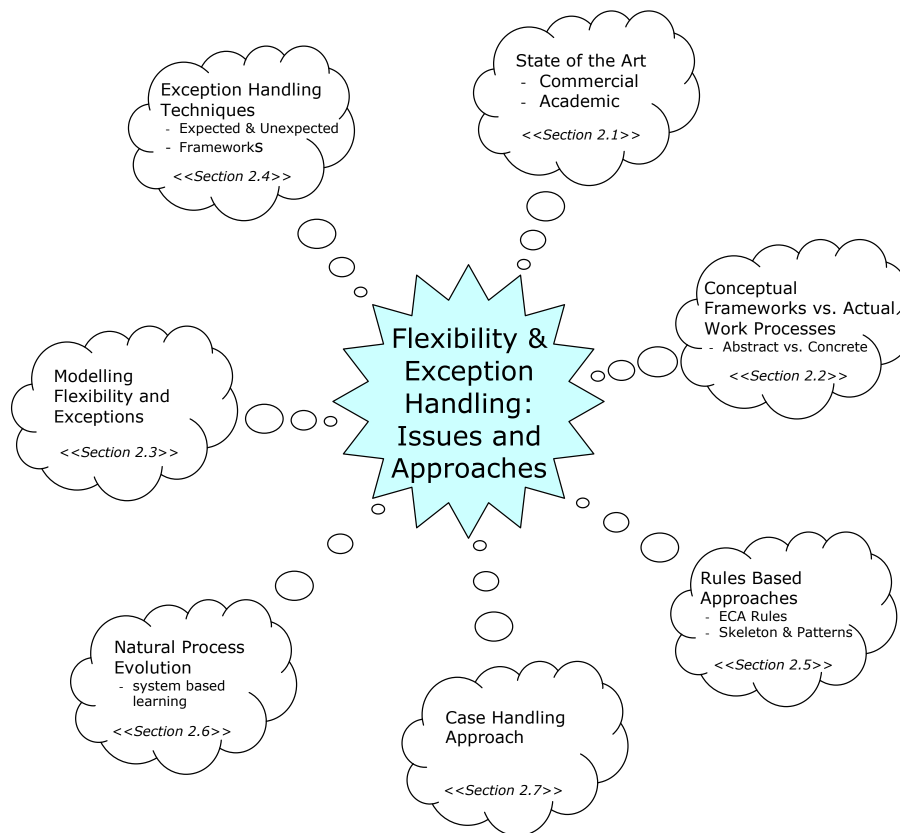

Figure 2.1: A Chapter Overview

the model statically. These typically flat, monolithic, single-schema architectures make it difficult to fully capture flexible business processes [83, 33]. While most provide solid support for transaction-level exceptions, there is very little support at the process-conceptual and instance-execution layers [87], besides triggers for deadlines being reached [152, 52] (A complete set of possible actions that a workflow engine should be able to accommodate when a deadline occurs is found in [68]). Even then, most simply inform a system administrator or other human stakeholder that a deadline has occurred — it is *expected* that the problem will be dealt with off-system.

So, while the ability to set deadlines is widely supported, what happens after a deadline expires is limited. Further, there is minimal support for handling: workitem failures (and even when that support is offered, they must be manually terminated); external triggers; and only one system reviewed (FLOWer) offers some constraint violation management [152].

All commercial WfMSs provide support for audit trails or logs. These logs, structured as either flat text files or database tables, allow for the recording of events that occur during the execution of a process, with various levels of granularity depending on the product. The logs

are made available for perusal by an administrator, either via a simple list dialog, or using a restricted set of available SQL querying constructs. There is no support for using those logs to dynamically evolve current workflow models based on processing history, nor is there any support for the system to dynamically interrogate those logs and make contextual choices regarding how the process can be executed, improved, or to provide possible exception handlers.

Tibco iProcess Suite (version 10.5) (formerly *Staffware*) [163] provides constructs called *event nodes*, from which a separate pre-defined exception handling path or sequence can be activated when an exception occurs at that point. It may also suspend a process either indefinitely or wait until a deadline occurs. If a workitem cannot be processed it is forwarded to a 'default exception queue' where it may be manually purged or re-submitted. A compensation workitem may be initiated when a deadline occurs. Also, a workitem may be externally triggered, or 'wait' until an external trigger occurs. Certain tasks may be manually skipped at runtime.

An optional component of the iProcess Suite is the *Process Orchestrator* [76], which provides for the dynamic allocation of sub-processes at runtime. It requires a construct called a "dynamic event" to be explicitly modelled that will execute a number of sub-processes listed in an 'array' when execution reaches that event. Which sub-processes execute depend on pre-defined data conditionals matching the current case. The listed sub-processes are statically defined, as are the conditionals. There is no scope for dynamically refining conditionals, nor adding sub-processes at runtime.

COSA (version 5.4) [59] 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, although models can be modified at runtime (but only for future instantiations). When a workitem fails the activity can be rolled back or restarted. A compensating activity can be triggered either externally or on deadline expiry. COSA also allows *manual* ad-hoc runtime adaptations such as reordering, skipping, repeating, postponing or terminating steps.

WebSphere MQ Workflow (version 6.0) [92] supports deadlines and, when they occur, will branch to a pre-defined exception path and/or send a notification message to a pre-defined user or administrator. Administrators can manually suspend, restart or terminate processes, or reallocate tasks. Only transaction-level exceptions are recognised, and they are simply recorded in the audit log.

SAP Workflow (version 6.20) [154] supports conditional branching, where a list of condi-

tions (each linked to a process branch) is parsed and the first evaluating to true is taken; all branches are pre-defined. Exception events are provided 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 — that is, filtering through a set of possible handlers based on the context of the case is not supported. When an exception occurs and a corresponding handler is found, all tasks in the block where the exception is caught are cancelled.

HP Process Manager (version 4.2) [86] allows administrators to manually intervene in process instances to handle external exceptions by initiating messages to executing processes. 'Detectors' for each exceptional event (including a manual command to abort the process) must be built as task-nodes into the standard model.

FLOWer (version 2.1) [32, 136], is described as a 'case-handling' system (cf. Section 2.7); the process model (or 'plan') describes only the preferred way of doing things and a variety of mechanisms are offered to allow users to deviate in a controlled manner [16]. 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 [152].

There have been a number of academic prototypes developed in the last decade (although activity was greater during the first half); very few have had any impact on the offerings of commercial systems [126]. Several of the more widely acknowledged are discussed here.

The *OPERA* prototype [82] 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 is found the entire process instance aborts.

The *eFlow* system [51] supports flexibility in e-Services by defining compensation rules for regions, although they are static and cannot be defined separately to the standard model. The system allows changes to be made to process models, but such changes introduce the common difficulties of migration, verification, consistency and state modifications.

ADEPT [142, 85] supports modification of a process during execution (i.e. add, delete and

change the sequence of tasks) both at the model (dynamic evolution) and instance levels (ad-hoc changes). Such changes are made to a traditional monolithic model and must be achieved via manual intervention, abstracted to a high level interaction. The system also supports forward and backward 'jumps' through a process instance, but only by authorised staff who instigate the skips manually [141].

The *ADOME* system [56] provides templates that can be used to build a workflow model, and provides some support for (manual) dynamic change; it uses a centralised control and coordination execution model to initiate problem solving agents to carry out assigned tasks. If a task raises an exception event, an Exception Manager will execute the exception handler defined for that task.

A catalog of 'skeleton' patterns that can be instantiated or specialised at design time is supported by the *WERDE* system [47]. Again, there is no scope for specialisation changes to be made at runtime.

AgentWork [127] 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 the approach detailed later in this thesis, and changes are limited to individual tasks, rather than the task-process-specification hierarchy. Also, the possibility exists for conflicting rules to generate incompatible actions, which requires manual intervention and resolution.

The *ActivityFlow* specification language described in [119] divides workflows into different types at design time (including ad-hoc, administrative or production), and provides an open architecture that supports interaction and collaboration of different workflow systems. While an open architecture is a useful concept and a desirable goal, the language does not provide any constructs for exception handling. Indeed, the paper advocates the use of a dedicated (human) workflow coordinator/administrator to monitor workflows with an eye on deadlines, handle exceptions, prioritise, stop, resume and abort processes, dynamically restructure running processes or change a specification. This list is useful as a checklist of qualities that the *automated* workflow management system developed by this research may be measured against.

The *TREX* system [159] allows the modeller to choose from a catalog of exception handlers during runtime to handle exceptions as they arise. However, as is the case with *ActivityFlow*, *TREX* 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.

Another approach to supporting dynamic change is presented by the *CoMo-Kit* system described in [62], which uses techniques sourced from knowledge based systems to present a model that concentrates, for the most part, on the data of a process rather than the control flow. While the model is presented at a relatively high level, it does provide some insight into how data can be modelled to better support dynamic change in workflows.

An interesting hybrid approach is called *XFolders* [53], a light-weight, document based workflow system that uses the metaphor of the inter-office memo circulation envelope to pass tasks from person to person in a work group following some loosely defined routes, until such time that the case is completed. *XFolders* provides flexibility by allowing: the dynamic addition of new users, new documents and new document repositories during a process instance; any person to take the decision to modify the process flow; and disconnected operations which allow for more flexibility for mobile users.

The *MARIFlow* system [64] has some similarities to *XFolders*, supporting document exchange and coordination across the internet. The system consists of a number of cooperating agents that are handle activities local to each site and coordinate the routing of documents between sites. 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 [171] 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 by the user. Thus users must be actively involved in the inference process during each case.

van der Aalst and Pesic point out that 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, and introduce a declarative approach to process modelling [138]. Their *DecSerFlow* graphical language [11] avoids many of the assumptions, constraints, conditions and rules that must be explicitly specified in procedural languages to perform flexible activities, the inclusion of which typically lead to an over-specification of the process. Relations between tasks (for example, that task A must follow task B at some time during process execution) can

be defined as hard constraints (which are enforced) and soft constraints (which may be violated) using the very concise and extendible modelling language.

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 solution introduced by this thesis avoids all of those issues.

2.2 Conceptual Frameworks vs. Actual Business Processes

One of the major focusses of this research was to gain a thorough understanding of how work is actually carried out in an organisational environment. A flexible workflow system that is fundamentally grounded in a well-understood description of work practice will better support that work, instead of the current offerings where the work performed is shoe-horned into the restrictions imposed by the manufacturing or production line metaphor used.

As more fully described in Chapter 3, this research takes the view that Activity Theory, a powerful and clarifying descriptive tool [130], can be employed as the foundation for a workflow management system that can offer better support to all work environments.

Workers require variety in their work processes [139]. This inherent human need to introduce variety into work is often incorrectly interpreted as exception producing. It can, rather, introduce *tension* between elements of a process which identify areas where systems no longer match the activities they model [57].

Rozinat and van der Aalst discuss the quantification of the differences between process models and real-world work processes in [151], and find that research in process mining reveals that there are often substantial differences between the real process and normative models of work processes. An example is the conformance of the testing process of machines which has a fitness of only 30% to the process logs accessed.

Suchman finds that work is not strictly governed by plans, but rather by the possibilities and limitations of the situation at hand [161]. Bardram [24] discusses how the idea of 'situated action', and how tightly a plan is adhered to during its execution, is often seen as opposed or

subsequential to actual planning work, but this does not necessarily have to be the case. He explores the connection between plans and their realisation in actual work. Most importantly, the author states that human knowledge about an environment is 'reflection based on activity', and that where actual activities digress from what was planned, a comparison is inherently made between the two and a consequent learning situation occurs.

In [19], this idea is further developed through a co-operative process model aimed at characterising work practices. It takes a phenomenological perspective which states that plans should not be seen as prescriptive: the deviation from a plan means the discovery of a plan that is inadequate; it does not equate to unauthorised or undesired behaviour. The paper makes use of Petri nets to model tasks, which ties in well with a graphical workflow specification approach. A case study using a related approach is found in [18] which discusses how workflow systems can be used, not only as passive modelers of work practices, but as proactive systems that provide support for the re-engineering of those practices towards efficiency gains.

Activity Theory is used by Kaasboll and Smordal to address issues of work context and how computers are used to mediate communication, co-ordination and interaction [103]. They use it as a basis for understanding the roles computers may play in an activity. Although the treatment is brief, and limited to the use of computers as tools, it does provide some theoretical basis that was extended by this research.

The shift of perspective of a computer's role in work environments from a means of control and administration to a mediation role is discussed in [157]. The author introduces an object oriented theoretical framework to address computers mediating collective activity. It splits computerised representations of work into two domains: *problem* (that part of the world that the system is controlling, for example a flight booking system); and *application* (the context of the work, for example the users, the organisation, ad-hoc activities, interruptions etc.). Smordal argues that while the problem domain is normally the one that is modelled, future models must also include the application domain in order to fully represent the work carried out and its environment. In other words, the *context* of each case must be taken into account to determine the specific execution of the workflow instance.

In [169], Twidale and Marty use a detailed case study of a (non-computer-based) system employed to process the relocation of a large museum and its exhibits. They use their observations to support the notion that collaborative systems are better suited to the business process environment they are attempting to emulate if the focus is more on error recovery than error

prevention. That is, in any human endeavour, it may be more productive to accept the fact that exceptions to any plan will occur in practice, and to implement support mechanisms which allow for those exceptional behaviours to be incorporated into the model, rather than to develop a closed system that tries to anticipate all possible exceptions, then fails to accommodate others that (inevitably) occur. This notion supports the idea of evolutionary workflow support systems, which over time and through experience tune themselves to the business process they are supporting.

Trajcevski, Baral and Lobo [165] introduce the idea of a workflow methodology that, in addition to allowing the expression of knowledge about a particular business process, allows the expression of 'ignorance', that is unknown values that reflect the realistic situation of processes that may have incomplete information about themselves at various execution times. Interestingly, they allow for 'sound' or 'mostly complete' specifications, which are then further developed dynamically during runtime as more information about the process becomes available.

The *ADEPT* system, described by Dadam, Reichert and Kuhn in [60], and by Reichert and Dadam in [144] and [143], is an attempt to move from functionally-based to process-based WfMSs. That is, the system recognises that business processes are abstract human creations, and a process-based system will better support the people doing the work, thereby delivering a system more capable of incorporating dynamic change. The articles use a medical scenario to outline how the system allows for changes made to executing instances of workflow processes, and gives important consideration to consistency. Their model employs a conceptual, graph-based approach that introduces a set of change operations designed to support dynamic change at runtime.

A taxonomy of change relevant to workflow systems is found in [10]. Six criteria for change are provided: What is the reason for change, what is the effect, which perspectives are affected, what kind of change, when are changes allowed and what to do with existing cases. The criteria can be applied to a workflow support system (including that resulting from this research) to measure the effectiveness of the system's ability to deal with change.

2.3 Modelling Dynamic Workflows

Many authors have researched the failure of WfMSs to successfully adapt to change, and have suggested possible treatments and/or solutions (for example [9, 49, 24, 109, 52, 54, 143, 60]).

A methodology for mapping expected exceptions on top of commercial WfMSs which do not directly support them is found in [52]. An analysis of a medical case study identified three main characteristics of expected exceptions: synchronicity (whether they occur during or between tasks), scope (specific to a single, or to multiple running process instances) and triggering event (categorised as data, temporal external or workflow events). This article also provides a valuable taxonomy of exception types.

An empirical study of call centre work practices in [17] shows that activities are completed by using a large set of informal routines that can be combined flexibly. In the study, management stated there is a constant need to find a balance between flexible diagnosis of caller needs with transactional efficiency, resulting in numerous minor task reallocations as the two demands were juggled while building a repertoire of small routines [17]. These results are strongly reflected in the solution offered by this research.

Malone et al. point out that current workflow systems can support only those processes that can be formalised in sufficient detail [122]. For more dynamic work environments to be successfully supported, focus must shift from building systems that execute processes to systems that *assist people* to carry out processes. Consequently, there is no requirement that all models be detailed or formalised to a degree sufficient for automation, but to describe processes at different levels of detail depending on particular needs and costs, negating the need to resolve difficult knowledge representation issues or investing a large amount in formalising organisational knowledge that is not immediately useful [122].

Problems created by the over-formalisation of work process models are discussed in [103]. It is claimed that it may lead to a de-skilling of workers (i.e. overspecialisation) and reiterates the complexities involved in formalising 'real-world' work activities. Tasks and authorisations often do not automatically align in sequence or schedule, but work is commonly driven by the situation at hand, often as a result of contingencies.

An argument for less structured routing mechanisms is found in [109], where it is stated that problems modelling dynamic work processes are due to the static routing constructs from one task to the next. While restricted in the article to form-based workflows, the idea of dynamic

routing between tasks based on context found applicability in the approach of this research.

Barthelmess and Wainer [30] make a clear distinction between the level of specification and the level of enactment; by doing so, they suggest that workflows may come to be seen as tools that allow organisations to effectively deal with exceptions, rather than viewing exceptions as failures in the workflow schema.

An approach that provided great direction for this research is called *Proclets* [5], which refers to light weight workflow processes that support interaction between multiple workflow instances at runtime. The paper discusses how WfMSs typically squeeze business processes into a single (complex) schema, when they would be better executed as separate entities with inter-schema messaging and communication protocols. This approach provides much insight into the modelling of the 'mini-processes' described by this research that can interact with a main 'parent' schema at runtime as required.

A method for the composition of processes at runtime is found in [63]. The authors propose a 'black box' from which activities may be manually selected by users at runtime in response to the needs of the particular case. While some constraint rules can be specified to prevent certain combinations of activities to be selected, there are no rules-based selections dependent on the actual case context — such decisions are made by each user for each case.

An interesting approach is found in [107] and [35], which discuss the modelling of work processes as a state-flow — that is, a process should be regarded as a trajectory within the space of all possible states. By starting with "full chaos", and then systematically adding constraints, it is argued that a state-flow model will be defined which allows for flexibility. The level of progress made towards the goal of the process can be determined by viewing the current process state (cf. Section 2.7). These articles provide insight into the way process states can be interrogated and used.

Workflow evolution and flexibility based on the concept of anticipation is proposed in [79]. This method involves using intermediate results of task executions as input into the selection of a path to subsequent tasks. A set of constraints is described for each task — when the constraints are met, the task is enabled. To achieve this end, normal graphical flow constraints are replaced by the definition of constraint sets for each task, which complicates the verification of the workflow.

2.4 Exception Handling Techniques

There are many examples in the literature of the failure of WfMSs to handle exceptions adequately, for example [49, 39, 114, 51, 46, 28].

Any software tool intended to support large, complex applications has to also support the pervasive problem of exceptions [81]. However, as discussed earlier, current business process and workflow systems provide, at best, minimal support for exception handling.

Eder and Liebhart [67] provided the now classic categorisation of workflow exceptions when they proposed that exceptional situations in WfMSs could be grouped into four sub-categories:

- **Basic failures**, which occur when there is a failure at the system level (e.g. DBMS, operating system, or network failure);
- **Application failures**, corresponding to failure of an application implementing a given task;
- **Expected exceptions**, which correspond to deviations from the normal behaviour of the process; and
- **Unexpected exceptions**, which occur when the corresponding workflow model does not properly model the actual business process activities.

Basic and application failures are generally handled at the system and application levels. WfMSs typically rely on the recovery mechanisms of the underlying operating systems and application interfaces, or the transactional properties of the underlying database platform to handle these kinds of failures, using methods such as retries and rollbacks. Since these types of failures occur outside of the conceptual workflow model, they are outside the scope of this research.

A clear and generic discourse on the fundamental makeup of workflow systems is given in [101]. Four key concepts (theory, technology, organisation and innovation) are discussed which are applicable to all workflow systems designed to model flexible workflows. A framework for handling unexpected exceptions is found in [41]. Although the handling mechanism uses human agents as 'on-line' exception handlers, it is claimed that the framework theoretically permits pre-programmed exception handlers.

Closely aligned to the topic of exception handling is that of supporting dynamic change, especially at runtime. In [1], van der Aalst brings together the concepts of supporting exception handling and dynamic change by proposing a generic process model approach to workflow management. He contends that exceptions are simply a special sub-set of occurrences that arise in an ad-hoc manner at runtime. Providing a system to handle these dynamic change events leads to the handling of exceptions and consequently workflow evolution.

Agostini and De Michelis argue that, in order to accommodate inevitable exceptions in workflow systems, the models that capture processes should be based on formal theory and be as simple as possible [20]. They should also support formal verification, graphical representations, multiple views and automatic changes to running instances. The developments discussed in this article had their genesis in [21], where a modelling system based on a subset of Elementary Net Systems was introduced, which allowed for simplicity and readability. These articles provide a good set of criteria against which a solution to the issues may be measured.

Dieters et al. [61] provide four basic techniques employed to deal with the occurrence of exceptions at runtime: basic build-ins (for transactional errors), late modelling (leave the detail until just before runtime), post modelling (monitor human interactions for later manual schema evolution) and off-line execution (perform tasks outside the workflow schema). All these techniques require costly human intervention at runtime, the last defeating the purpose of workflows entirely. Obviously, to minimise intervention and downtime during workflow execution is a major goal of any organisation, and therefore a primary focus of this research.

In [49], the difficulties faced when modelling exceptions in commercial workflow engines are discussed. Generally, exceptions must be built into the standard workflow, which can quickly complicate even the simplest of cases. This problem is also explored by Horn and Jablonski [89]; using a large market research organisation as a case study, they claim that up to 30 different possible (though unlikely) execution paths can exist at any one point in a typical process, all of which must be explicitly modelled using current commercial techniques. The approach presented in this thesis achieves the graphical description of exception handlers without the overheads inherent in building all exceptional cases explicitly into the standard model.

Building an object-oriented workflow system based on event handling is described in [164]. An interesting approach (so far as this research was concerned) is the development of reusable workflow components called "event occurrence brokers" which can be called asynchronously to react to certain events. There is some correlation between this concept and the triggering

mechanism for exception events (both synchronous and asynchronous) used by this research.

Mourão and Atunes [125, 124] argue that user involvement is unavoidable when determining how to handle an exception, and propose a system which uses a workflow process to collaboratively guide users through an exception handling decision-making procedure for another workflow process. While this approach has merit in orchestrating a "solution-by-committee", an over-reliance on user input appears to be counter-productive in terms of efficiently completing processes.

2.5 Rules-Based Approaches

Fabio Casati and his colleagues are prolific and respected authors with respect to workflow management systems and more specifically the treatment of exceptions. They provide a thorough treatment of workflow exceptions in the various papers relating to their *WIDE* workflow model. In [46], the author discusses Eder and Liebhart's taxonomy of workflow exceptions mentioned previously.

In [47], Casati observes that expected exceptions can fall into one of four categories:

- **Workflow exceptions**: which can only occur at the beginning or completion of tasks, that is while the workflow is in transition from one task to the next, and can therefore be regarded as synchronous.
- **Data exceptions**: caused by attempts to change workflow data structures to an inconsistent state.
- **Temporal exceptions**: raised at certain timestamps or intervals (eg deadlines) which act much like interrupts to the running workflow; and
- **External exceptions**: which are raised when the workflow is notified of occurrence of certain external events that will affect the process flow.

Casati then argues that, since the majority of exceptions are asynchronous (i.e. all besides workflow exceptions), and may not be raised by task completions, they cannot be suitably represented within a graph of tasks, and that therefore the only way to handle exceptions is programmatically via event-condition-action (ECA) rules. This is the approach taken in the *WIDE*

prototype, which results in a workflow specification where exception handlers cannot be modularised (i.e. cannot be separated from the standard workflow model). Although Baresi et al. [28] claim this to be an advantage, since the ECA rules for exceptions are stored separately from the graphical representation of the workflow, it still enforces two different paradigms for modelling workflows.

A more detailed treatment of exception handling using ECA rules in the WIDE model is found in [51], and [50] provides a detailed description of how the language developed for the *WIDE* model, Chimera-Exec, is used to define exception rules. The former paper also discusses other WfMSs (such as *ObjectFlow*) that have limited exception handling mechanisms which result in the active workflow being aborted when the handler is executed. Casati et al. argue that the WIDE ECA approach allows normal flow to proceed while the exception is handled. A similar approach (but with a different rule paradigm) based on programming and process models is taken by Borgida and Murata in [39], and in more detail in [40].

An ECA rules approach to exception handling is also employed in the *ADOME WfMS*, described in [56] and [55], and in more technical detail in [54]. The ADOME system utilises workflow 'templates' that can be used as a basis for a workflow at design time, providing some support for (manual) dynamic change. Chiu et al. provide an excellent taxonomy of workflow exception sources, types and handlers.

The ECA approach is extended in [120] by augmenting the condition with a *justification* so that both must evaluate to true for the corresponding action to be taken — this approach is labelled *defeasible workflow*. The authors present a framework where context dependent reasoning is used (via the justification parameter) to enhance the exception handling capability of workflow systems, together with a case-based reasoning 'mechanism' requiring human involvement and consideration of previous cases to resolve handling approaches for particular case instances.

In [48], and [47], Casati et al. provide an interesting analysis of workflow exceptions, and also describe how they may sometimes demonstrate a generic behaviour that may be abstracted as a 'skeleton' which can be instantiated and/or specialised by the modeller at design time. These papers describe the *WERDE* environment, which consists of a catalog of skeleton patterns that the modeller has at his or her disposal. These patterns, in keeping with the *WIDE* ideology, are stored and used as textual ECA rules. However, they are only available at design time, which means the value of such a system is limited to making the design process a little easier.

On the other hand, the *WERDE* environment does afford some ideas regarding the cataloguing of schemas that were beneficial in framing some of the outcomes of this research.

The *Liaison* workflow architecture, described in [117], also provides an interesting framework from which to build. It treats roles, agents, schedules and data as separate entities and proposes a basis for forming relationships between them. Although the architecture allows off-line workflow evolution only and the paper generally lacks detail, the hierarchy outlined presents a novel view of how a WfMS may be implemented.

Another interesting approach is the *SWORDIES* project discussed in [71], which proposes a system of structured rules representing a business process at different abstraction levels. The rules-based approach is used to build an 'integration layer' into which different modelling constructs can be transformed or translated into a generic set of rules which are in turn translated by a rule-oriented workflow engine layer for use in a variety of different workflow engines. This approach has some similarities to the orthogonal structure of the service constructed in this research, where the implementation is independent of the underlying workflow engine.

2.6 Automatic Evolution of Workflow Specifications

Activity Theory offers relevant descriptors for how learning evolves in 'learning systems' [139]. Each learning system must have: the ability to make plans and action them; a memory or data storage; the ability to evaluate their own behaviour with access to storage; and the ability to incorporate that evaluation into future plans and actions [139]. Figure 2.2 [139] shows Rauterberg's model of an organic learning system (which also applies to groups and organisations), which points to a metaphor for a computer-based learning system that assists workflow process evolution and exception handling procedures.

It is argued in [140] that the conventional understanding of human error as a negative event to be avoided is an artefact of Western culture. We are constrained from consideration of faults and weaknesses, even though it is through them that we learn experientially. The authors contend that, from a purely introspective viewpoint, human behaviour cannot be erroneous, but represents the best expectation into the near future fulfilling all actual constraints.

Boardman proposes a system which uses 'Activity Footprints' that comprise of two levels: the high-level activity space and its associated information spaces [36]. These activity footprints

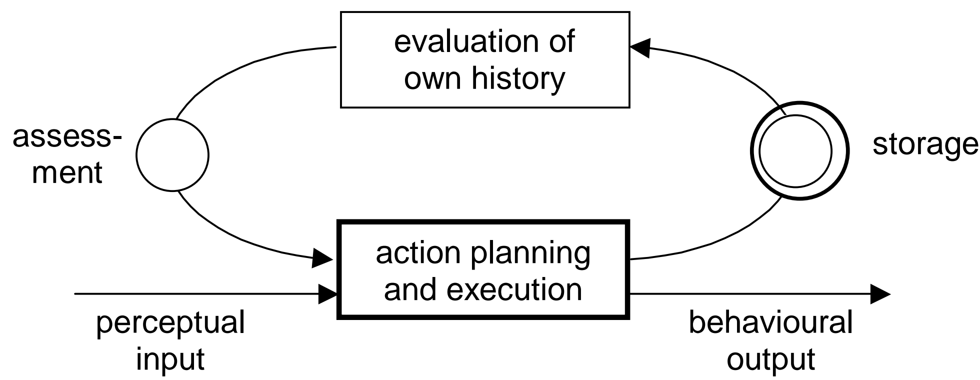

Figure 2.2: Components of a learning system (from [139])

can be re-used, each time with further refinements taking place, thereby supporting learnt and collaborative evolution.

An algorithm for extracting workflow process models from logs of unstructured past executions is given in [22]. The directed graph constructed by the algorithm represents the control flow of the process, and satisfies completeness, redundancy and simplicity requirements. It also addresses the issue of noise in the log, that is, the collection of erroneous data. While the algorithm provides complete process models, and depends on previously successfully executed processes, it does provide a useful insight into how new processes can be derived from existing historical data. Further current extensions into this area are elaborated in [2], [14] and [8] which provide valuable grounding in the mining of workflow logs and other archival information for use in evolving workflow models.

A knowledge-based approach, using previous problem solving cases to provide insight into possible solutions for new problems via human intervention is outlined in [43]. Although the article is light on detail, a useful framework is described called a 'survivability architecture', which divides the architecture into four levels: instance, schema, workflow and infrastructure. At the first two levels, adaptation and exception handling methods are available, but require user intervention. However, the interventions are recorded so that they may be used to assist future decision making.

The *WAM* workflow system [73] archives all the events that occur during a workflow execution. Events, which include such things as creating and completing tasks and process models, are categorised and weighted. From the set of events, a *Causality Tree* is composed, which gives insight to the actual work situation and allows better management of cases. While this use of archival data is limited, it does provide a lead in how historical data can be applied to workflow

processes.

2.7 Case Handling

A relatively recent development in workflow technologies has been the introduction of Case Handling Systems (CHS). The case handling paradigm seeks to overcome the limitations of rigidity inherent in workflow systems by using a data-driven approach.

Case Handling is a response to the lack of flexibility perceived as inherent in typical WfMSs. It is considered that the rigidity imposed by WfMSs is a consequence of work routing being the only driver of the process, which leads to these problems [16]:

- Work needs to be *straight-jacketed* into activities — although tasks are considered atomic in a workflow process, actual work is considered to be carried out at a much finer grained level;
- Routing is used for both work distribution and authorisation, thus only crude mechanisms can be used to align the process to the organisational hierarchy;
- Focus on control flow leads to context being marginalised — workers only know that part of the process that they directly perform; and
- Routing focuses on what should be done instead of what can be done, resulting in inflexible workflows.

Case handling takes the approach that the manufacturing paradigm of most WfMSs does not meld with the way work is carried out in most non-manufacturing environments [7]. Case handling attacks this in two ways: each participant is provided with a broader view of the entire process (thus avoiding *Context Tunnelling*, where participants have knowledge only of their own tasks), and purports better support for exceptions [149, 145].

An example of a commercial CHS is the FLOWer system, which employs a series of forms to capture and present data pertaining to a case. A process design in FLOWer consists of four elements: a process flow, to determine processing sequence; a role graph, to record roles and interrelationships; data which describes the case; and forms [32, 136].

While the goals of case handling systems and workflow systems are similar, the properties of case handling differ in three characteristics (from [145]):

- **Focus is on the case:** In WfMSs, a worker will be presented with a very specific work item related to a task that is defined within a pre-defined process model. Thus, the worker has no real knowledge of how the work item relates to the overall activity (or case). In CHSs, the entire case is presented to the worker, who can browse and/or update the case with relevant information. Further, the worker does not need to wait for the case to be submitted to them; it is always available to those with necessary access rights [145].
- **Process is data-driven:** WfMSs manage process instances as progress through the process definition, and produces two distinct types of information: logistic (which is used to manage instance execution) and content (which is stored exclusively in external applications) [145]. Conversely, in CHSs the state of the case is the complete set of data, which is fully managed by the system. This makes it possible at the data level to review the progress of an activity, thus countering the flexibility limitations of traditional WfMSs.
- **Implicit Modelling:** The process definition of traditional WfMSs is a set of precedence relations between an enumeration of tasks. Any routing between tasks not specified during the definition of the process will not be allowed to occur at execution. In CHSs, only a preferred or 'normal' flow is modelled [145]. At runtime, a user may diverge from the flow definition (assuming proper authorisations) by re-doing, skipping or rolling back tasks. It is claimed that this implicit modelling produces models that, when an exception occurs, will allow a user to select the appropriate tasks to perform. Such models should also be easier to interpret by users.

However, it can be argued that CHSs do not provide the 'magic bullet' claimed, since:

- It has not been proven that context tunnelling is a problem [145].
- In keeping a complete set of data related to a case within the CHS, in addition to the data stored in the external applications used, a great deal of redundant, and potentially inconsistent data, may be maintained. This counteracts the primary goal of WfMSs: to improve efficiencies by dividing logistics from content.

- Having the actual value of case data drive the state of the case invites potentially inconsistent situations. Since several workers may have access to the entire case, it may happen that two may want to update the same data contemporaneously. Also, several workers having the ability to re-do, skip or rollback tasks within the same case may lead to inconsistencies. Conversely, prohibiting simultaneous access to a single case rules out concurrency, which is naturally supported by traditional WfMSs.

- Re-do, skip and rollback primitives can also be expressed in traditional WfMSs, therefore CHSs do not offer more flexibility at runtime than WfMSs [145].

- In a CHS, workers will need to spend more of their time on task co-ordination (since they have complete access to the case at all times during its life), than with a WfMS (which explicitly co-ordinates tasks).

- In a CHS model, workers may be granted the ability to re-do a task or set of tasks infinitely, which contradicts efficient process execution, and may even allow the task to never reach completion [145].

The case handling paradigm is an interesting approach which has several commonalities with Activity Theory. For example, it provides support for the concept of the activity as the basic unit of analysis. By allowing access to the complete case by workers, limited by access permissions, it also supports work communities and interrelationships, and divisions of labour. It provides for process flexibility that better captures what can be done, instead of following a rigidly routed process explicitly. It also considers exceptions as a natural and necessary part of a work process.

In these areas, the approach reflects that of this research, to the extent described above. It is interesting that Activity Theory closely describes their approach, although the CHS authors make no reference to Activity Theory in their discussions.

Conclusion

Drawing on the review of literature in this chapter, it is evident that while there have been many approaches to the issues of flexibility and exception handling in workflows, particularly among the academic prototypes, no one approach seems to provide solutions to all of the key

problem areas raised. One possible reason is that many are built as extensions to an underlying framework that imposes rigidity in process expression and enactment.

Such approaches usually involve a reliance on manual intervention and runtime user interactivity. Thus, they are in direct opposition to the perceived gains offered by workflow systems. Users are forced to incorporate ‘unnatural’ actions into their work routines so that their work processes match that of the workflow specification, or take their work off-system until such time as the process model can be made to more closely reflect their actual work practices, which involves complex update, modification and migration phases.

It is clear that before a solution can be considered, the fundamental base of workflow support systems, that is work practices they aim to support must be examined. By first determining how people actually work, rather than relying on artificial constructs, it is feasible that a workflow system can be built that better reflects real work processes, and therefore provides a consistent support mechanism for all work environments.

Chapter 3

Theoretical Framework: Activity Theory

This chapter discusses, in some detail, the limitations of the computational or production-line metaphor in regards to its use in conceptualising workflow processes. It then provides a summary of Activity Theory, which forms the theoretical framework for the outcomes described in the remainder of this thesis.

A set of principles applicable to capturing work practices using a workflow process model are derived, then mapped to a series of criteria (based on those discussed in the previous chapters) against which a workflow solution based on Activity Theory may be measured.

3.1 Issues in Representing Work Practices Procedurally

Whenever a series of actions is undertaken with a view to achieving a pre-conceived result, some plan or set of principles is implemented that guide and shape those actions towards that goal. To be effective, a plan must be described using constructs and language that are relevant to both the actions being performed and the desired result, and be comprehensible by its participants and stakeholders. In workflow terms, analysts seek to model some aspect of the real world by using a metaphor that bears some resemblance to the real world, but also represents an understanding of computational processes. Such metaphors are abstract constructions that form a common reference model which assist us in representing the external world through computers.

The fundamental and widely understood *computational* metaphor [158] takes a set of inputs, performs a series of functional steps in a strict sequence, and, on completion, produces some output that represents the goal of the process. Thus the computational metaphor describes

a single, centralised thread of control, which very much reflects its mathematical ancestry but also reveals an underlying misconception in the common perception of technological ‘progress’. Technological developments are, according to Holt, “as much affected by fashion as clothing” [88]. Technologies do not evolve automatically (as Marx assumed) but rather reflect prevailing human culture [128]. That is, new technologies are derived from perceived needs and realised, not in isolation, but through the conventions and norms of their social *milieu*.

Thus the traditional computational metaphor reveals the influence of pioneers such as von Neumann and his team, and especially Turing, whose abstract machine proposed ’step-at-a-time’ processing [167], and which in turn reflects the influence on thinking of the contemporaneous development of assembly-line manufacturing [84].

As the prevailing technological advances influenced the structure of early computers, so too has the computational metaphor become a significant model system for the conceptualisation and interpretation of complex phenomena, from cognition to economics to ecology [158]. Of particular interest is the way the metaphor has been applied to the definition of organisational behaviour issues and the representation of organisational work processes. The computational metaphor remains applicable to well-defined problem domains where goal-directed, sequential, endpoint-driven planning is required [158]. Such domains were the early beneficiaries of workflow management systems. Consequently, current commercial workflow systems provide support for standardised, repetitive activities that do not vary between execution instances.

Adherence to the metaphor by WfMSs has been an important factor in their acceptance by organisations with structured work practices. Descriptions can be found throughout the workflow literature to the ’processing’, ’manufacturing’ and ’assembly-line’ modelling metaphors that are employed by commercial workflow systems. However, while the Workflow Management Coalition claims that “even office procedures can be processed in an assembly line” [173], there are many aspects where administrative and service processes differ from manufacturing processes [7].

The prescriptive and rigid nature of workflow systems do not allow for unanticipated variations [40]. But in many environments, business processes vary greatly, are dynamic with regards to their actual execution [143], and may differ between individual instances. Even for highly structured work practices (such as banking and air-traffic control), it remains difficult (if not impossible) to successfully capture all work activities, and in particular all of the task sequences possible, in a standard workflow model, at least not without producing some very

complex models. It may be that the computational metaphor has become an inhibiting factor in the development of workflow systems able to effectively support flexible work practices.

Many technologically adept systems fail because they ignore human and social factors [155, 132]. It is therefore desirable to extend the capabilities of WfMSs, so that the benefits offered to organisations employing rigidly defined, 'production-line' processes could also be enjoyed by those businesses which employ more creative or flexible processes. A new approach is needed which better incorporates the fluidity between a computer, its user and their environment. However, before such an approach can be realised, a thorough understanding of the network of interrelationships between workers, goals, tools and environment must be built.

A workflow management system that better supports flexible work environments requires a sound theoretical foundation that describes how work is conceived, carried out and reflected upon. One such theoretical base can be found in Activity Theory.

3.2 An Introduction to Activity Theory

3.2.1 Background

Activity Theory is a powerful and clarifying descriptive tool rather than a strongly predictive theory, and incorporates notions of intentionality, history, mediation, collaboration and development, focussing on understanding everyday practice in the real world [130].

Activity Theory originated in the former Soviet Union in the 1930's and 40's as part of the cultural-historical school of psychology founded by Vygotsky and extended by Leontiev. It takes a humanist perspective to the analysis of activity, and human activity is the fundamental unit of that analysis.

In the form presented by Leontiev [115], Activity Theory subsequently became one of the major Soviet psychology theories, and was used in areas such as the education of disabled children and the ergonomic design of equipment control panels.

In the 1980s, Activity Theory came to the attention of Scandinavian information technology researchers (for example: [130], [110], [37] and [38]). Their contribution was a revised formulation of Activity Theory, and also the application of Activity Theory to human-computer interface design.

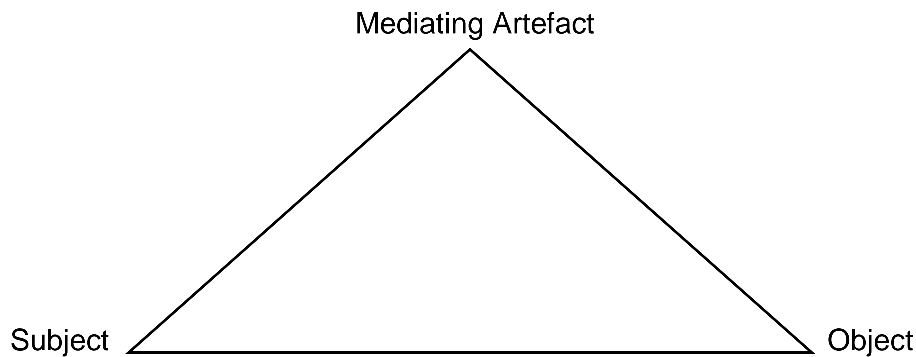

Figure 3.1: Vygotsky's Model [170]

3.2.2 How Activity Theory Describes Work Practices

Activity Theory was first formulated by the Soviet psychologist Lev Vygotsky [170], who began working on the theory at a time when the prevailing dominant psychological theories were based on reflexology, which attempted to reduce all psychological phenomena to a series of stimulus-response chains.

Vygotsky's first challenge to established concepts was to suggest that individual consciousness is built from the outside through relations with others, and higher mental functions must be viewed as products of mediated activity. The role of mediator is played by psychological tools (such as plans, experiences, concepts and reflections) and means of interpersonal communication.

When Vygotsky formulated his first ideas about the mediation of consciousness, he appropriated Marxist ideas about how tools or instruments mediate work activities. For Marx, labour is the basic form of human activity — in carrying out labour activities, humans do not simply transform nature: they themselves are also transformed in the process. New types of tools are created to carry out the continually evolving forms of labour activity. Conversely, each new level of tools gives rise to yet another round of ways of conceptualising and acting on the world [172].

To summarise: (i) people carrying out work procedures invent new tools to allow them to work more efficiently, and (ii) whenever a new tool is introduced, people discover new ways of conceiving and carrying out their activities.

As opposed to the study of the individual as a separate entity, in Activity Theory the unit of analysis is an activity. An activity is composed of a subject, and an object, mediated by a tool (Figure 3.1) [170]. A subject is a person or a group engaged in an activity. An object (in the

sense of 'objective') is held by the subject and motivates activity, giving it a specific direction. The mediation can occur through the use of many different types of tools, material tools as well as mental tools, including culture, ways of thinking and language. Activity Theory considers computers as special kinds of tools mediating human interaction in the world. Computers are used not because people want to interact with them but because they want to reach their goal beyond the situation of the 'dialog' with the computer [106].

People consciously and deliberately generate activities in part through their own objectives. Activity is both internal to people (involving specific goals) and, at the same time, external to people (involving artefacts, other people, specific settings). The crucial point is that in Activity Theory, external and internal are fused together. An activity is a form of *doing* directed at an object. Transforming the object into an outcome motivates the existence of the activity. An object can be a material thing, less tangible (a plan) or totally intangible (a common idea), as long as it can be shared for manipulation and transformation by the activity participants [110].

In Vygotsky's early work there was no recognition of the interplay between participants; Alexei Leontiev extended the theory by adding several features based on the need to separate individual action from collective activity. The hierarchical distinction between *activity*, *action* and *operation* was added to delineate an individual's behaviour from the collective activity system.

The following example (from Leontiev [115]), classically describes the various aspects of Activity Theory:

> A beater, taking part in a primeval collective hunt, was stimulated by a need for food or clothing, which the flesh and skin of the dead animal would meet for him. However, his activity was directed at frightening a herd of animals and sending them toward other hunters, hiding in ambush. That, properly speaking, is what should be the result of the activity of this man. And the activity of this individual member of the hunt ends with that. The rest is completed by the other members. This result does not in itself lead to satisfaction of the beater's need for food. Thus the outcome or objective of his activity did not coincide with the motive of his activity; the two are divided. Processes, the object and motive of which do not coincide with one another are called *actions*. We can say, for example, that the beater's activity is the hunt, and the frightening of the game his action.

> The beater who needs food for survival is engaged in actions that result in the opposite of what he is immediately seeking. Instead of closing the distance with the quarry, he is driving it away. This makes sense only if he knows that someone is waiting to achieve his goal (consciously shared with others) at the other end. The sense of his action was not in the action itself but in his relation to other members of the group.

It is apparent from the example that more than one action can be used to achieve a goal, both the beaters and the hunters in the activity system above are carrying out actions which will result in a successful hunt. But their actions are different.

Therefore, actions cannot be considered in isolation; they are impossible to understand without considering their situation within a context. Thus, a minimal meaningful context for individual actions must be included in the basic unit of analysis — that is, an activity [110]. However, there are no firm borders between an activity and an action. For example, a software project may be an activity for the development team, but the company director may see each project as an action in his activity of managing the firm [110].

The third hierarchical level which Leontiev added to the theory of activity was the level of operations, which are tasks performed automatically (or unconsciously) dependant on the context of each activity.

As a result of the need to consider the shared meaning of activity, the initial theory was reconfigured by Yrjö Engeström the addition of rules, community and the division of labour, and renamed the Activity System (Figure 3.2) [72]. In this model of an Activity System:

- **Subject** refers to the individual or group whose point of view is the focus of the model involved in the activity.
- **Object** (or objective) is the target of the activity.
- **Instruments** refer to internal or external mediating artefacts (or tools) which help to achieve the outcomes of the activity.
- **Community** is comprised of one or more people who share the objective with the subject.
- **Rules** regulate actions and interactions within the activity system.

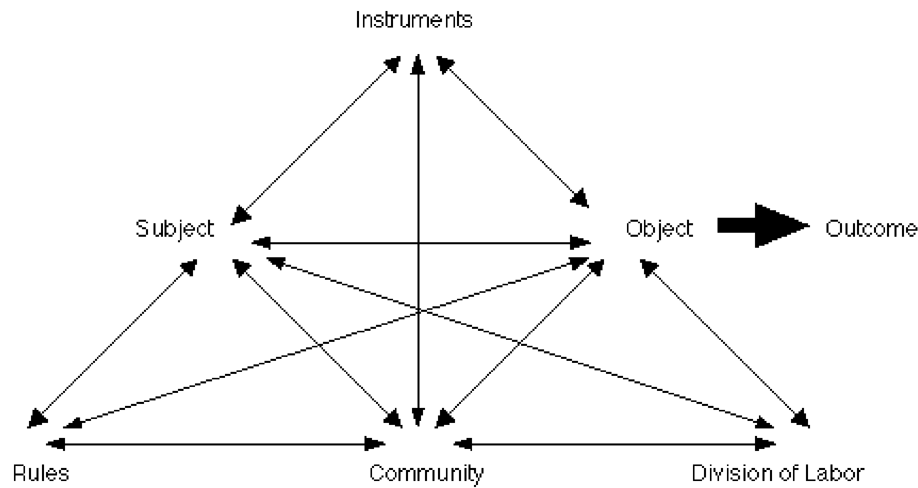

Figure 3.2: The Activity System [72]

- **Division of Labour** informs how tasks are divided horizontally between community members as well as referring to any vertical division of power and status.

Activities are not static entities; they are under continuous change and development, which is not linear or straightforward but uneven and discontinuous. This means that each activity has a history of its own. Parts of older versions of activities stay embedded in them as they develop, and historical analysis of the development is often needed to understand the current situation [110].

Activities are not isolated units but are influenced by other activities and environmental changes — these influences may cause imbalances. Activity Theory uses the term *contradiction* [72] to indicate a misfit between elements, within them, and between different activities. Contradictions manifest themselves as problems, ruptures, breakdowns, clashes :- *exceptions*. Tension that arises between elements of the Activity System identify areas where systems no longer match the activities they model [57]. Activity Theory sees contradictions not as problems but as sources of development; activities are virtually always in the process of working through contradictions that subsequently facilitate change [130].

Before an activity is performed in the real world, it is typically planned using a model. Generally, the better the model, the more successful the activity. However, models and plans are not rigid and accurate descriptions of the execution steps but always incomplete and tentative [110]. Artefacts themselves (which include plans and models) have been created and transformed during the development of the activity itself and carry with them a particular culture — a historical residue of that development [130].

Plans can be used as historical data by investigating not adherence to the plan, but the deviations from it. It is the deviations that represent a learning situation, and therefore only the deviations need to be recorded. Future plan modifications can be achieved by studying and/or incorporating the deviations [153]. Activities are always changing and developing. Change takes place at all levels; new operations form from previous actions as skill levels increase, the scopes of actions modulate and totally new actions are invented, experimented with, and adapted as responses to new situations or possibilities [110].

According to Leontiev [116], in principle all operations can be automated. By automating and substituting human operations, IT can become part of the activity and expand the scope of actions available to participants. IT can also support actions in various ways. For a workflow management system to accomplish its goals, it must contain a computerised representation of the structure of the work environments procedures and activities. Traditionally, these representations have been linear or sequential or hierarchical decomposition of activities into tasks and are built separately to the execution of the task. Such models cannot take into account unexpected events and have trouble dealing with exceptions. But unexpected events, or deviations, are not exceptions but are a natural and very important part of the activity, and provide the basis for learning and thus evolving the plan for future instantiations.

The experience of using a plan to guide an activity is gained during the instantiation of the activity. In order for plans to become resources for future instantiations of an activity it is important that the planing tool allows for the ongoing creation and dynamic modification of a plan based on experience gained while operating the plan.

One of the traditional limitations of workflow implementations in less than strictly defined work processes is that it is very difficult if not impossible to incorporate every deviation into the workflow template and therefore future instantiations of the plan. Some deviations may apply to every future instance. Some may only apply once in a while but these cases should not be left out of the plan. In the normal course of events, these ‘deviations’ are performed externally to the system. But with the workflow structure put forward by this research, incorporating dynamic change is a fundamental feature of its design.

To summarise, Activity Theory states that human activity has four basic characteristics [24]:

1. Every activity is directed towards a material or ideal object satisfying a need, which forms the overall motive of the activity. For example, a physician's medical treatment of a

patient is directed toward curing the patient.

2. Every activity is mediated by artefacts, either external (a stethoscope) or internal (cognitive — using medical concepts, knowledge and experience).

3. Each individual activity is almost always part of collective activities, structured according to the work practice in which they take place. For example, a patient diagnosis can seldom be established without reference to a diversity of medical information, which is obtained by consulting with specialised personnel at different service departments such as radiology, laboratory and pathology. Thus collective activities are organised according to a division of labour.

4. Finally, human activity can be described as a hierarchy with three levels: *activities* realised through chains of *actions*, which may be performed through *operations*:

   - An activity consists of one or more actions, and describes the overall objective or goal.
   - An action equates to a single task carried out to achieve some pre-conceived result. For example, a medical treatment activity may consist of five actions: (i) examination, (ii) requesting and interpreting information, (iii) diagnosis, (iv) further treatment and (v) monitoring. Each action is achieved through operations determined by the actual conditions in the context of the activity.
   - Operations describe the actual performance of the action, and are dependant on the context, or conditions that exist for each action. Exactly how the action of examination is performed, for example, depends clearly on the concrete conditions, e.g. the type of disease in question, the state of the patient, available medical instruments, knowledge and experience of the physician, and so on.

3.3 Applicability of Activity Theory to Workflow Systems

Figure 3.3, taken from [72], shows a reworking of the Activity System incorporating both the dynamic interrelationships between subject, object and community, and the various roles computers may play in a collective activity. The interrelations are [157]:

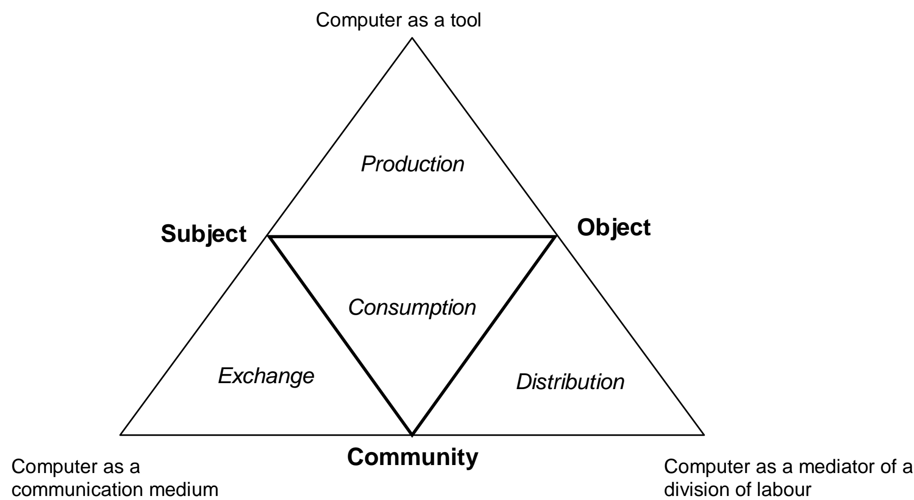

Figure 3.3: Aspects of Collective Activity [72]

- *Production* shows the relationship between the subject and the object, mediated by tools.
- *Distribution* shows the relationship between the community e.g. the other employees in an work team and the object, mediated through the division of labour (for task coordination, distribution and authorisation).
- *Exchange* shows the relationship between the subject and the community, mediated through communication and accepted social behaviours.
- *Consumption* shows the use of the products and/or services of the activity, and represents an external view (how customers use the services of the enterprise).

A flexible workflow system, based on Activity Theory, would simultaneously provide support for all points of the triangular model, and provide the ability to construct a process model that embraces the entire diagram [129]. In addition, Table 3.1, taken from [110], states in general terms how Activity Theory can support information technology activities. The worklet service coming out of this research provides support for all of these intersecting levels.

3.4 Principles Derived from Activity Theory

The value of any theory is not whether it provides an objective representation of reality [27], but rather how well it can shape an object of study, highlighting relevant issues [29]. Ten fundamental principles, representing an interpretation of the central themes of Activity Theory applicable

Table 3.1: A classification of potential ways of supporting activities by information technology [110]

	Operation level support	Action level support	Activity level support
Tool	Automating routines	Supporting manipulative actions; making tools and procedures visible and comprehensible	Enabling the automation of a new routine or construction of a new tool
Object	Providing data about an object	Making an object manipulable	Enabling something to become a common object.
Actor	Triggering predetermined responses	Supporting sense-making actions	Supporting learning and reflection with respect to the whole activity
Rules	Embedding and imposing a certain set of rules	Making the set of rules visible and comprehensible	Enabling the negotiation of new rules
Community	Creating an implicit community by linking work tasks of several people together	Supporting communicative actions; making the network of actors visible	Enabling the formation of a new community
Division of Labour	Embedding and imposing a certain division of labour	Making the work organisation visible and comprehensible	Enabling the reorganisation of the division of labour

to an understanding of organisational work practices, have been derived by this research and are summarised below. These principles will be referred to as p1...p10 in Section 3.5.

- **Principle 1: Activities are hierarchical** An activity consists of one or more actions. Each action consists of one or more operations.
- **Principle 2: Activities are communal** An activity almost always involves a community of participants working towards a common objective.
- **Principle 3: Activities are contextual** Contextual conditions and circumstances deeply affect the way the objective is achieved in any activity.
- **Principle 4: Activities are dynamic** Activities are never static but evolve asynchronously, and historical analysis is often needed to understand the current context of the activity.
- **Principle 5: Activities are mediated** An activity is mediated by tools, rules and divisions of labour.
- **Principle 6: Actions are chosen contextually** A repertoire of actions and operations is created, maintained and made available to any activity, which may be performed by making contextual choices from the repertoire.

- **Principle 7: Actions are understood contextually** The immediate goal of an action may not be identical to the objective of the activity of which the action is a component. It is enough to have an understanding of the overall objective of the activity to motivate successful execution of an action.

- **Principle 8: Plans guide work** A plan is not a blueprint or prescription of work to be performed, but merely a guide which is modified depending on context during the execution of the work.

- **Principle 9: Exceptions have value** Exceptions are merely deviations from a pre-conceived plan. Deviations will occur with almost every execution of the plan, and give rise to a learning experience which can then be incorporated into future executions.

- **Principle 10: Granularity based on perspective** A particular piece of work might be an activity or an action depending on the perspective of the viewer.

3.5 Functionality Criteria Based on Derived Principles

This section discusses six functionality criteria that a WfMS would need to meet to support the principles of work practice derived from Activity Theory listed in Section 3.4. Each criterion broadly represents a major topic area of workflow research. Each is discussed with reference to the relevant Activity Theory principles (see Table 3.2 for a summary mapping), and how well it is supported by some leading commercial workflow products, and some research prototypes.

3.5.1 Criterion 1: Flexibility and Re-use

Workflow systems should support the utilisation of actions contextually chosen from a repertoire, which can be employed within several different activities. The analysis of human activity as a three-level hierarchical structure emphasises that the same activity can be realised through different actions and the same action can be realised through different operations [24]. Conversely, the same action can be a component of several different activities *(p1, p5)*. While some workplaces have strict operating procedures because of the work they do (for example air traffic control) many workplaces successfully complete activities by developing a set of informal routines that can be flexibly and dynamically combined to solve a large range of problems [17].

Table 3.2: Summary mapping of Activity Theory principles vs. workflow functionality criteria

	Flexibility & Re-use	Adaptation via Reflection	Dynamic Evolution	Locality of Change	Comprehensibility of Models	Exceptions as 'First-Class Citizens'
Activities are Hierarchical	✓			✓	✓	
Activities are Communal			✓			
Activities are Contextual	✓	✓				
Activities are Dynamic		✓	✓	✓		
Mediation of Activity	✓	✓	✓			
Actions are Chosen Contextually	✓					✓
Actions are Understood Contextually			✓	✓		
Plans Guide Work		✓			✓	✓
Exceptions have Value		✓	✓			✓
Granularity Based on Perspective					✓	

Thus, realising an activity in a contingent environment is aided considerably by having a repertoire of actions and operations to choose from [24] *(p6)*. This denotes a crucial factor for the representation of flexible workflows. At any point in time, there may be several possible sequences that can be followed utilising a sub-set of available actions to achieve the objective of the activity. Choices are made dependent on the actual circumstances of the activity at that point in time *(p3)*. Thus a flexible WfMS would manage a catalog of actions that at runtime could ideally be chosen programmatically, but in any case with minimal human intervention, as contextual information presents itself. Since each action has it own goal, each should be a distinct, self-contained workflow process in its own right.

The availability of a catalog of actions would also encourage re-use, since actions may be made available for use in several distinct activities, and/or used as templates for the definition of new actions. By using a modular approach, a resultant model may range from a simple skeleton to which actions can be added at runtime (supporting dynamic adaptation) or may be a fully developed model representing the *a priori* complete work practice, depending on user and organisational needs.

The commercial products reviewed all provide modelling frameworks that are basically monolithic, but with various levels of support for the deconstruction of tasks. *iProcess* provides 're-usable process segments' that can be inserted into any process. *HP Process Manager* provides some support for sub-processing, but not as distinct workflows. *SAP R/3* allows for the definition of 'blocks' that can be inserted into other 'blocks', thus providing some support for encapsulation and reuse. *COSA* supports parent-sibling processes, where data can be passed to/from a process to a sub-process. The *ADOME* system [56, 55, 54] 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 [48, 47]. No product or prototype provides an extensible repertoire of processes from which a dynamic selection can be made at runtime.

3.5.2 Criterion 2: Adaptation via Reflection

Workflow systems should support evolutionary adaptation of processes based on the experience gained during each execution of the process [69]. All human activity is guided by the anticipation of fulfilling the objective of that activity. This anticipation is formed as a result of

reflection on experiential memory [25]. Thus, a work plan is the result of reflection on prior experiences and in anticipation of a particular goal. Because of this, plans are not rigid and accurate descriptions of the execution path but always incomplete and tentative [110] *(p5, p8)*. An instantiation of a plan is fundamentally distinct from the plan itself. A plan is a work resource [155], detached from concrete activities, and is used to organise and divide the work amongst the participants involved.

To achieve the expected result, the actions and operations contained in the plan (conceptual) have to be adjusted to the material conditions of each situation (contextual) [162]. Depending on the context of the instantiation, some actions will mirror the plan, while other parts of the plan may be either discarded or augmented in some way. After completion, a comparison is made between the anticipated and actual outcomes, and any incongruities or deviations from the plan add to the experience of the participants, and so give rise to a learning situation *(p3, p4, p8)*. Plan adaptations for future instantiations can be achieved by recording the occurrence of deviations and incorporating them as required [153]. Thus a plan is an artefact that contains historical residue of its development [130] *(p5, p8)*.

Typically, workflow systems ignore these deviations, resulting in systems that remain static in the face of change. This leads to increasing instances of work being performed off-system to accommodate those changes [160]. A dynamic, flexible, evolving workflow support system must have the ability to record deviations from the plan, why the deviation occurred and how it was handled, thus handling the deviation on-system. In addition, the 'handlers' thereafter simply become an implicit part of the plan (that is, a possible path) for future instantiations *(p9)*. Therefore, it is important that the workflow system allows for the ongoing evolution and dynamic modification of a model based on experience gained while operating an instantiation of it. Also, where a deviation was unexpected, a guided choice might be made available to the designer, allowing for the implementation of an appropriate action chosen from the catalog. A designer might be guided by the system to choose certain actions over others based on system 'intelligence' using archival data and heuristic algorithms.

All the commercial products reviewed require the model to be fully defined before it can be instantiated, and changes must be incorporated by modifying the model statically. They provide little support for learning from past instantiations besides keeping basic audit logs of executed events in a database table, with a list facility available for manual perusal. *ChangeEngine* and *iProcess* also provide monitoring systems which allow administrators to observe current

executions, view statistics and be informed when certain milestones are reached. In the research field, the *Milano* system aims to provide process models as 'resources for action' rather than strict blueprints of work practices [19]. The *ADEPT* system is process-based, rather than functionally-based, and is designed to support dynamic change at runtime [60, 144, 143]. The *WAM* system [73] archives all events during execution, which are subsequently weighted and categorised. A 'causality tree' is then composed which provides insights into how the work was carried out. Approaches to extracting improved models from logs of previous executions are proposed in [22, 15, 8, 14].

3.5.3 Criterion 3: Dynamic Evolution of Tasks

Workflow systems should support the evolution of processes towards individual specialisations without risking the loss of motivation for the overall activity. Almost all work practices are collective activities that involve cooperation and mutual dependence between participants *(p2)*. Even so, practices generally evolve towards minimising mutual articulation among individuals, without jeopardising the overall objective of the collective activity *(p4)*. Activity Theory recognises this as a strict division of labour, enabling participants to specialise in certain actions, and by designing certain structured artefacts that encapsulate actions in an efficient way [24] *(p5)*. However, over-specialisation of an action may lead to deskilling, and therefore loss of motivation, of the workers involved [103]. This phenomenon, referred to as context-tunnelling [149], becomes an issue when a participant is no longer motivated by the objective of the activity *(p7)*.

Workflows structured as an assembly line are often the cause of such situations, because they force workers to become mere processors in a product delivery chain — that is, the human contextual factors are ignored. The manifestation of the inherent need to introduce variety into work processes [139] is often incorrectly interpreted as exception producing *(p9)*. Activity Theory recognises that there is no need for each participant to know the detail of the entire activity (as asserted by proponents of case handling systems [145]), but only the context of their own action in relation to the whole activity. Since a participant is motivated by the objective of an activity, only an understanding of how the goal of their own action contributes to the overall objective, and the freedom to seek efficiencies and/or variety in achieving that goal, is required *(p7)*.

FLOWer [32, 136] addresses the context-tunnelling problem by capturing and presenting

data pertaining to the entire case via a series of forms to all authorised participants. All the remaining reviewed commercial products present to each participant only that information they require to complete their pre-defined task.

3.5.4 Criterion 4: Locality of Change

Modifications should be able to be fully applied by changing a minimal number of components, and should impact minimally on associated components. Related to support for flexibility is support for locality of change. Bass, Clement and Kazman [31] suggest that increasing the adaptability of software is largely a function of how localised necessary modifications can be made. In terms of workflow process models, this idea suggests two desirable and related goals. Firstly, to ensure that a workflow process model is strongly adaptable, modifications should be able to be limited to as few components as possible. Secondly, any changes made within those components should impact on other components to the least extent possible.

Activity Theory describes the focus for change as the individual participant who conceptualises improvements in methods at the action level *(p4, p7)*. Therefore, a workflow support system can achieve adaptability by strongly providing for locality of change at that level. One approach is to support the definition of a workflow process as a set of sub-processes, each of which is a distinct entity representing a single action *(p1)*. Also, since each sub-process will be fully encapsulated, any changes made to one will not impact other sub-processes available to the same model. This is not achievable with monolithic modelling paradigms.

All the commercial products reviewed require modifications to be applied to the whole model — that is, change is not a trivial exercise for models of any complexity. Also, no research prototypes were found that supported the idea of sub-processes being distinct workflows in their own right.

3.5.5 Criterion 5: Comprehensibility of Process Models

Process models should be comprehensible to all stakeholders, and should support representation at different levels of granularity. A major limiting factor in the uptake of workflow systems is the complexity of the models developed for all but the most trivial activities. WfMSs that require the development of monolithic models fail to take into account the complexities in adapting

and evolving those models. Also, there may be stakeholders that have difficulty in interpreting models that contain many possible paths on a single plane, and/or require different levels of abstraction to be presented *(p1, p10)*.

While the modeller will have a detailed understanding of the components and structure of the model, a workplace manager will generally not have the expertise to decipher it, nor will a staff member charged with performing a particular action *(p8)*. However, the very acceptance of a workflow solution may hinge on the ability to provide management with a modelled representation of a work process which is comprehensible to them [45]. An approach which may offer better understanding is the representation of workflow models as a hierarchical set of linked sub-processes, since each sub-process would be fully encapsulated and therefore a (simpler) workflow model in its own right. This approach would also allow for the model to be represented at different levels of granularity for different stakeholders.

While the various abilities of the analysed commercial products to support sub-processes have been stated above, none have the ability to present models of differing granularity to stakeholders. One research prototype that does is *SWORDIES* [71], which uses rules to represent work processes at different abstraction levels that are built into an 'integration layer' at runtime.

3.5.6 Criterion 6: The Elevation of Exceptions to "First-Class Citizens"

Workflow exceptions should not be regarded as errors but as events that provide an opportunity for a learning experience and therefore are a valuable part of any work practice. Generally, commercial WfMSs take the view that exceptions are to be considered errors. As such, they are seen to be annoyances which either should have been foreseen and therefore prevented from occurring in the first place, or perceived as impossible to predict and therefore best left handled off-system. However, Activity Theory regards exceptions (contradictions, diversions, tensions, deviations) as a normal and valuable part of every work activity, which can be used to refine or evolve the process towards improved efficiencies. As a result, exceptions should be implemented in a positive fashion to better reflect (within the workflow model) the actuality of the work practice it supports *(p9)*.

The ability to capture exceptional events and deal with them appropriately in real time, while concomitantly using that exception as a learning experience to evolve the workflow, would be

a central requirement of any workflow system based on Activity Theory. An effective system must also provide the ability to source an appropriate exception 'handler', then dynamically incorporate it into the running instance. That is, exception handling techniques should be available during both the design and execution phases. The principle of contextual choice from a repertoire of actions can also be applied as an appropriate mechanism for exception handling *(p6, p9)*. If, based on a set of conditions, contexts, archival data and so on, a matching handling sub-process could be found for a particular event, then that process could be automatically invoked. If no such match could be found, then a list of possible matches, selected from a catalog of processes by applying heuristic techniques, might be presented to an administrator from which an appropriate choice could be made or a new handler created using one of the returned matches as template. The extraction of general knowledge of workflow events using knowledge discovery methods based on archival data, conditions and context reflects the 'learning system' central to Activity Theory *(p8, p9)*.

Although there has been much research in the field of workflow exceptions, few results have been incorporated by commercial workflow products [126, 14, 52]. Currently, there is only trivial support amongst them for exception handling. *iProcess* provides constructs called event nodes, which suspend execution until the occurrence of a pre-defined exception, at which time an exception handling path or sequence is activated. *COSA* provides support for timeout exceptions only, at which time notification is sent to an administrator. *ChangeEngine* allows administrators to manually intervene in process instances to handle exceptional events by initiating messages to executing processes — 'detectors' for each exceptional event (including a manual command to abort the process) must be built as nodes into the standard model. *WebSphere* provides no concept of exceptions as expected or unexpected events at the conceptual or process execution layers besides simple deadline, which when triggered sends a message to an administrator. *SAP R/3* provides for pre-defined branches which are built into the standard monolithic model. Exception events are provided 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 of each type to each task. The *OPERA* prototype [81] allows for exceptions to be handled at the task level, where control is handed over to a single handler predefined for the type of exception that has occurred, or propagated up various ancestor levels throughout the

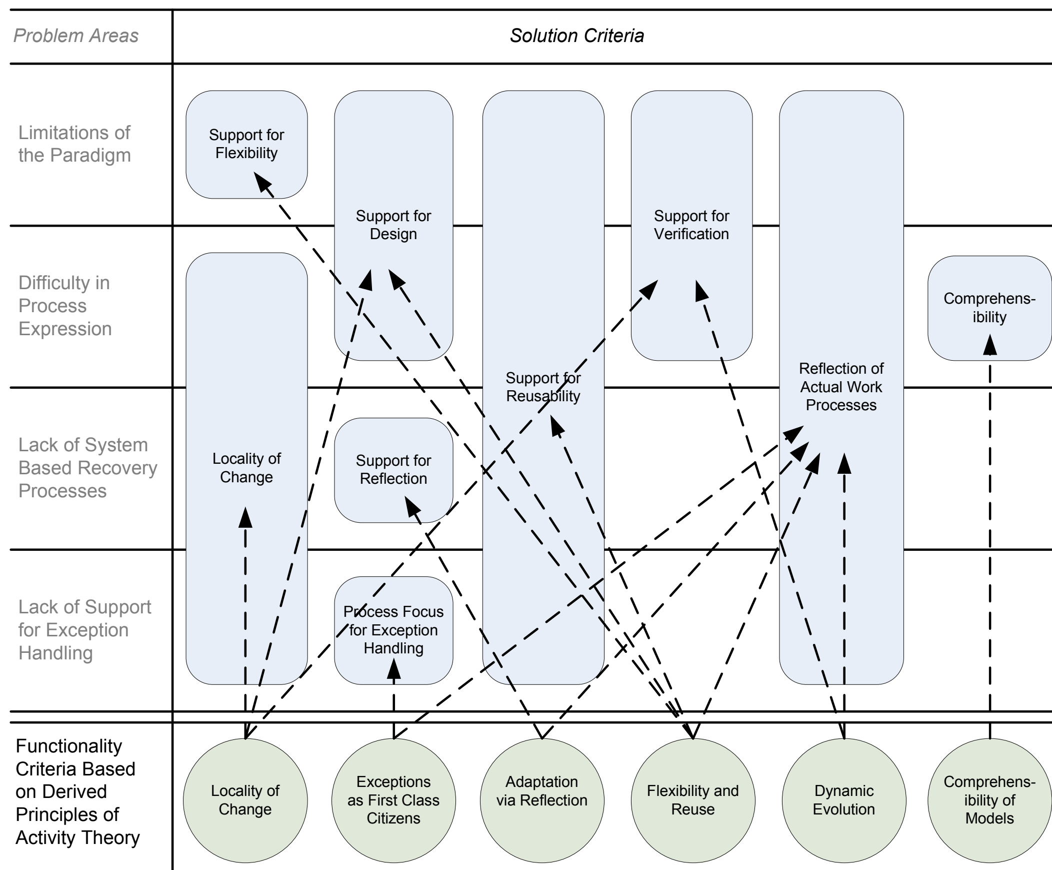

Figure 3.4: Relating Derived Functionality Criteria to the Solution Criteria of this Research

running instance. The *eFlow* system [51] uses rules to define exceptions, although they cannot be defined separately to the standard model.

3.6 Functionality Criteria vs. Solution Criteria

In Section 1.3, nine solution criteria were defined as those that would need to be satisfied to successfully meet the challenges posed by the four key problem areas addressed by this research. Figure 1.2 showed a mapping of the criteria to the problem areas.

The six functionality criteria emerging from the principles derived from Activity Theory in this chapter bear much in common with the nine solution criteria. Figure 3.4 shows an updated version of Figure 1.2 where the relationships between the solution criteria and the functionality criteria are mapped. It can be seen that some have a one-to-one relationship, while other functionality criteria map across several of the solution criteria.

It is clear from the summary relationships of Figure 3.4 that the principles derived from

Activity Theory do indeed reflect the actual requirements for a workflow system that better supports flexibility and exception handling.

Conclusion

This chapter provides an extrapolation of the applicability of Activity Theory to the implementation of more flexible and better directed workflow support and management systems. Recent work in IT research in relation to Activity Theory has been undertaken in the areas of Human-Computer Interface design [37, 38, 110, 106, 130], Artificial Intelligence [84, 139, 140, 158], Cooperative Learning [95, 112] and Software Analysis and Design [26, 24, 38, 57, 78, 129, 168]. However, there has been very little work done on the direct relationship of Activity Theory to workflow management.

Activity Theory offers a number of interesting insights into workflow research domains, particularly the related issues of workflow adaptability, flexibility, evolution and exception handling. The principles derived in this chapter have been applied to the implementation and deployment of the system described in the following chapters and will hopefully provide further guidance to others in this field. While the commercial products reviewed support few of the principles derived from Activity Theory, and research prototypes support some individually, there are no systems (outside of this research) that meet most or all of the principles.

Chapter 4

Conceptualisation of Worklets

In the previous chapter, a set of principles was derived from Activity Theory and applied to the issues of flexibility and exception handling for workflow systems. From that mapping of principles to issues, it was found that:

1. Workflow management systems typically have trouble supporting all but the most rigid business processes precisely because their frameworks are based on computing metaphors rather than accepted ideas of actual work practices.
2. A workflow management system that sought to overcome those issues must be built around a framework that better mirrors the way people perform work activities in organisational environments.

The consideration of these findings formed the conceptual foundations of a discrete service that transforms otherwise static workflow processes into fully flexible and dynamically extensible process instances that are also supported by dynamic exception handling capabilities. That service has been named the *Worklet Service*.

This chapter represents a conceptual view of the Worklet Service. It should be read in conjunction with, and as a prelude to, the two chapters following which describe in detail the formalisation and implementation of the service respectively.

4.1 Worklets — An Introduction

Fundamentally, a workflow management system that is based on the principles derived from Activity Theory would satisfy the following criteria:

- *A flexible modelling framework* — a process model is to be regarded as a guide to an activity's objective, rather than a prescription for it;
- *A repertoire of actions* — extensible at any time, the repertoire would be made available for each task during each execution instance of a process model;
- *Dynamic, contextual choice* — to be made dynamically from the repertoire at runtime by considering the specific context of the executing instance; and
- *Dynamic process evolution* — allow the repertoire to be dynamically extended at runtime, thus providing support for 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.

Thus, to accommodate flexibility, such a system would provide each task of a process instance with the ability to be linked to an extensible repertoire of actions, one of which to be contextually and dynamically chosen at runtime to carry out the task. To accommodate exception handling, such a system would provide an extensible repertoire of exception-handling processes to each process instance, members of which to be contextually and dynamically chosen to handle exceptions as they occur.

An outcome of this research has been the implementation and deployment of such a system, created using a service-oriented architecture. To support flexibility, the service presents the repertoire-member selection actions as *worklets*. In effect, a worklet is a small, self-contained, complete workflow process which handles one specific task (action) in a larger, composite process (activity).[1] A top-level or parent process model is developed that describes the workflow at a macro level. From that manager process, worklets may be contextually selected and invoked from the repertoire of each enabled task, using an associated extensible set of selection rules, when the task instance becomes enabled during execution. New worklets for handling a

[1]In Activity Theory terms, a worklet may represent one action within an activity, or may represent an entire activity.

task may be added to the repertoire at any time (even during process execution) as different approaches to completing a task are developed, derived from the context of each process instance. Importantly, the new worklet becomes an implicit part of the process model for all current and future instantiations, avoiding issues of migration and version control [6, 3, 108, 99]. In this way, the process model undergoes a dynamic natural evolution.

In addition, for each anticipated exception (an event that is not expected to occur in most instances), a set of repertoire-member exception handling processes, known as *exlets* (which may include worklets as compensation processes) may be defined for handling the event, to be dynamically incorporated into a running workflow instance on an as-needed basis (see Section 4.4). That is, for any exception that may occur at the task, case instance or specification level, a repertoire of exlets may be provided, the most appropriate one system-selected at runtime based on the context of the case and the type of exception that has occurred. Further, worklets that are invoked as compensation processes as part of an exception handling process are constructed *in exactly the same way* as those created to support flexibility, which in turn are constructed in the same way as ordinary, static process models.

In the occurrence of an unanticipated exception (i.e. an event for which a handling exlet has not yet been defined), then either an existing exlet can be manually selected (re-used) from the repertoire, one may be adapted on the fly to handle the immediate situation, or a new exlet constructed and immediately deployed, in each case allowing execution of the process instance that raised the exception to take the necessary action and either continue unhindered, or, if specified in the exception handler, to terminate, as required. Crucially, the method used to handle the new exception and a record of its context are captured by the system and immediately become an implicit part of the parent process model, and so a history of the event and the method used to handle it is recorded for future instantiations.

The worklet approach provides support for the modelling, analysis and enactment of business processes, and directly provides for dynamic exception handling, ad-hoc change and process evolution, without having to resort to off-system intervention and/or system downtime.

4.2 Context, Rules and Worklet Selection

For any situation, there are multiple situational and personal factors that combine to influence a choice of action. That set of factors that are deemed to be *relevant* to the current situation we call its *context*.

The consideration of context plays a crucial role in many diverse domains, including philosophy, pragmatics, semantics, cognitive psychology and artificial intelligence [42]. Capturing situated context involves quantifying and recording the relevant influencing factors and relationships between the inner state and the external environment [147].

A taxonomy of contextual data that may be recorded and applied to a workflow instance may be categorised as follows (examples are drawn from a medical treatment process):

- **Generic (case independent):** data attributes that can be considered likely to occur within any process (of course, the data values change from case to case). Such data would include descriptors such as when created, created by, times invoked, last invoked, current status; and role or agent descriptors such as experience, skills, rank, history with this process and/or task and so on. Process execution states and process log data also belong to this category.
- **Case dependent with *a priori* knowledge:** that set of data that are known to be pertinent to a particular case when it is instantiated. Generally, this data set reflects the data variables of a particular process instance. Examples are: patient name and id, blood pressure readings, height, weight, symptoms and so on; deadlines both approaching and expired; and diagnoses, treatments and prescribed medications.
- **Case dependent with no *a priori* knowledge:** that set of data that only becomes known when the case is active and deviations from the known process occur. Examples in this category may include complications that arise in a patient's condition after triage, allergies to certain medications and so on.

Methods for capturing contextual propositions typically focus on collecting a complete set of knowledge from an 'expert' and representing it in a computationally suitable way [105]. Such approaches depend heavily on the expert's ability to interpret their own expertise and express it in non-abstract forms [123]. However, experts often have difficulty providing information

on how they reach a specific judgment, and will offer a justification instead of an explanation [105]. Furthermore, the justification given varies with the context in which the judgement was made.

Bouquet et al. [42] place current theories of context into two distinct groups: *divide-and-conquer*, a top-down approach that views context as a way of partitioning a global model into simpler pieces (for example the ‘expert’ approach described above), and *compose-and-conquer*, a bottom-up approach that holds that there is no tangible global model to begin with, but only local perspectives, and so views context in terms of locality in a (possible or potential) network of relations with other local perspectives.

A top-down, complete representation of knowledge within a given domain is considered by many researchers to be impossible to achieve in practice, and is perhaps not even desirable [42]. In terms of using context as a factor in computational decision making, it is considered more judicious to capture only that subset of the complete contextual state of a particular domain relevant to making a correct and informed decision.

One bottom-up approach to the capture of contextual data that offers an alternative method to global knowledge construction is *Ripple Down Rules* (RDR), which comprise a hierarchical set of rules with associated exceptions, first devised by Compton and Jansen [58].

The fundamental feature of RDR is that it avoids the difficulties inherent in attempting to pre-compile a systematic understanding, organisation and assembly of all knowledge in a particular domain. The RDR method is well established and fully formalised [156] and has been implemented as the basis for a variety of commercial applications, including systems for reporting DNA test results, environmental testing, intelligent document retrieval, fraud detection based on patterns of behaviour, personal information management and data mining of large and complex data sets [135].

An RDR Knowledge Base is a collection of simple rules of the form “if *condition* then *conclusion*” (together with other associated descriptors), conceptually arranged in a binary tree structure (for example, Figure 4.1). Each rule node may have a false (‘or’) branch and/or a true (‘exception’) branch to another rule node, except for the root node, which contains a default rule and can have a true branch only. If a rule is satisfied, the true branch is taken and the associated rule is evaluated; if it is not satisfied, the false branch is taken and its rule evaluated [65]. When a terminal node is reached, if its rule is satisfied, then its conclusion is taken; if its rule is not

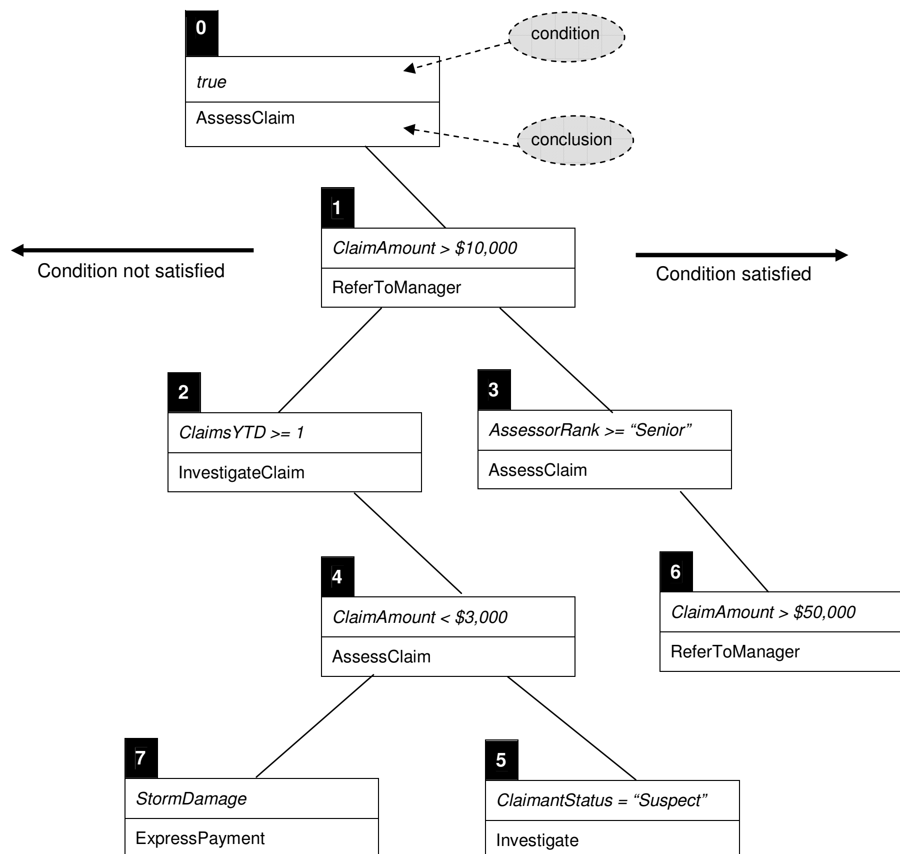

Figure 4.1: Conceptual Structure of a Ripple Down Rule (*Assess Claim* Example)

satisfied, then the conclusion of the last rule satisfied on the path to that node is taken. For terminal nodes on a true branch, if its rule is not satisfied then the last rule satisfied will always be that of its parent (since it must have evaluated to true for the terminal node to be evaluated).

This tree traversal gives RDR implied *locality* — a rule on an exception branch is tested for applicability only if its parent (next-general) rule is also applicable. This feature provides the fundamental benefit of RDR: general rules are defined first, and refinements to those rules added later as the need arises or as knowledge about the domain grows [156]. Thus, there is always a working rule set that extends over time.

For example, the rule tree in Figure 4.1 represents the following illustrative set of (somewhat artificial) business rules:

1. **Node 0:** By default, assess a claim;

2. **Node 1:** An exception to the node 0 rule is that for those cases where the claim amount is greater than $10,000, the claim must be referred to a manager (since the condition of node 0 is always satisfied, the condition of node 1 will always be evaluated);

3. **Node 3:** An exception to the node 1 rule is that cases where the claim amount is greater than $10,000 may be assessed by an assessor of at least 'Senior' ranking (satisfied node 1);

4. **Node 6:** An exception to the node 3 rule is that those cases where the claim amount is greater than $50,000 must always be referred to a manager (regardless of the rank of available assessors; satisfied node 3);

5. **Node 2:** If the claim amount is less than $10,000 (and so the condition of node 1 was not satisfied), then if the claimant has already made a claim in the current year it must be investigated;

6. **Node 4:** An exception to the node 2 rule is that if the claim amount is also less than $3,000, simply assess the claim (i.e. satisfied node 2, but the amount is too trivial to warrant an investigation);

7. **Node 5:** An exception to the node 4 rule is that for those cases where the claimant status is set to 'suspect', the claim should always be investigated (satisfied node 4);

8. **Node 7:** If the claim amount is less than or equal to $10,000 (unsatisfied node 1) and there has been a claim from this claimant in the current year (satisfied node 2) and the claim amount is also greater than or equal to $3,000 (unsatisfied node 4) and the claim is for storm damage, then the claim should be escalated for express payment.

It should be noted here that it has been empirically shown that Ripple Down Rules are able to describe complex knowledge systems using less rules than conventional 'flat' rule lists [156, 74, 58].

If the conclusion returned is found to be unsuitable for a particular case instance — that is, while the conclusion was correct based on the current rule set, the circumstances of the case instance make the conclusion an inappropriate choice — a new rule is formulated that defines the contextual circumstances of the instance and is added as a new leaf node using the following algorithm:

- If the conclusion returned was that of a satisfied terminal rule, then the new rule is added as a local exception to the exception 'chain' via a new true branch from the terminal node.

- If the conclusion returned was that of a non-terminal, ancestor node (that is, the condition of the terminal rule was not satisfied), then the new rule is added via a new false branch from the unsatisfied terminal node.

In essence, each added exception rule is a refinement of its parent rule. This method of defining new rules allows the construction and maintenance of the rule set by "sub-domain" experts (i.e. those who understand and carry out the work they are responsible for) without regard to any engineering or programming assistance or skill [105].

Each node incorporates a set of case descriptors, called the 'cornerstone case', which describe the actual case that was the catalyst for the creation of the rule. When a new rule is added to the rule set, its condition is determined by comparing the descriptors of the current case to those of the cornerstone case and identifying a sub-set of differences. Not all differences will be relevant — it is only necessary to determine the factor or factors that make it necessary to handle the current case in a different fashion to the cornerstone case to define a new rule. The identified differences are expressed as attribute-value pairs, using the normal conditional operators. The current case descriptors become the cornerstone case for the newly formulated rule; its condition is formed by the identified attribute-value pairs and represents the context of the case instance that caused the addition of the rule.

Rather than impose the need for a closed knowledge base that must be completely constructed *a priori*, this method allows for the identification of that part of the universe of discourse that differentiates a particular case *as the need arises*. Indeed, the only context of interest is that needed for differentiation, so that rule sets evolve dynamically, from general to specific, through experience gained as they are applied.

Ripple-Down Rules are well suited to the worklet and exlet selection processes, since they:

- provide a method for capturing relevant, localised contextual data;
- provide a hierarchical structuring of contextual rules;
- do not require the top-down construction of a global knowledge base of the particular domain prior to implementation;
- explicitly provide for the definition of exceptions at a local level;
- do not require expert knowledge engineers for its maintenance; and

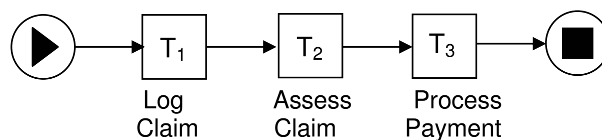

Figure 4.2: Simple Insurance Claim Model

- allow a rule set to evolve and grow, thus providing support for a dynamic learning system.

Each worklet is a representation of a particular situated action that relies on the relevant context of each case instance, derived from case data and other (archival) sources, to determine whether it is invoked to fulfil a task in preference to another worklet within the repertoire. When a new rule is added, a worker describes the contextual conditions as a natural part of the work they perform[2]. This level of human involvement — at the 'coalface', as it occurs — greatly simplifies the capturing of contextual data. Thus RDR allows the construction of an evolving, highly tailored local knowledge base about a business process.

4.3 The Selection Process

The worklet approach allows for two related but discrete areas of dynamic and flexible workflow to be addressed: dynamic selection of tasks, and exception handling with corrective and compensatory action. A conceptual synopsis of the selection process is dealt with in this section; exception handling in the next.

Consider the simple insurance claim example in Figure 4.2[3]. When this specification is created, one or more of its tasks may each be linked to a corresponding repertoire of worklets from which one will be substituted for the task at runtime. Each task associated with the worklet service has its own particular repertoire, and the members of which may be found in a number of other repertoires. Along with the specification, a corresponding RDR rule set is created which defines the conditions to be evaluated against the contextual data of the case instance. That is, each task may correspond to a particular 'tree' of RDR nodes within which are referenced a repertoire of worklets, one of which may be selected and assigned as a substitute for the task dynamically. Not all tasks need be linked to a repertoire — only those for which worklet

[2]In practice, the worker's contextual description would be passed to an administrator, who would add the new rule.

[3]All of the process models in this thesis are defined using YAWL notation. See Figure 6.1 for a description of the notation elements.

substitution at runtime is desired.

Each task that is associated with a worklet repertoire is said to be 'worklet-enabled'. This means that a process may contain both worklet-enabled tasks and non-worklet-enabled (or ordinary) tasks. Any process instance that contains a worklet-enabled task will become the parent process instance for any worklets invoked from it.

Importantly, a worklet-enabled task remains a valid (ordinary) task definition, rather than being considered as a vacant 'placeholder' for some other activity (i.e. a worklet). The distinction is crucial because, if an appropriate worklet for a worklet-enabled task cannot be found at runtime (based on the context of the case and the rule set associated with the task), the task is allowed to run as an 'ordinary' task, as it normally would in a process instance. So, instead of the parent process being conceived as a template schema or as a container for a set of placeholders, it is to be considered as a complete process containing one or more worklet-enabled tasks, each of which *may* be contextually and dynamically substituted at runtime.

It is possible to build for a task an initial RDR rule tree containing many nodes, each containing a reference to a worklet that will be used if the conditional expression for that node and its parent nodes are satisfied; alternately, an initial rule tree can be created that contains a root node only, so that the worklet-enabled task runs as an ordinary task until such time that the rule tree is extended to capture new contextual scenarios (which may not have been known when the process was first defined). Thus, the worklet approach supports the full spectrum of business processes, from the highly structured to the highly unstructured.

Referring again to the example (Figure 4.2), suppose that, after a while, a new business rule is formulated which states that when a claim comes to be assessed, if the claim amount is more than $10,000 then it must be referred to a manager. In conventional workflow systems, this would require a re-definition of the model. Using the worklet approach, it simply requires a new worklet to be added to the repertoire for the *Assess Claim* task and a new rule added as a refinement to the appropriate RDR by the administrator. That is, the new business rule is added as a localised refinement of a more general rule (see Figure 4.1).

The modified RDR tree can be used to extrapolate a view or schematic representation of the model, with the modified rule for the *Assess Claim* represented as XOR choice (Figure 4.3[4]). That is, a tool can be used to translate the RDR set back into a *view* of a set of tasks and con-

[4]In YAWL notation, T_1 represents an XOR-split task, T_3 and XOR-join task.

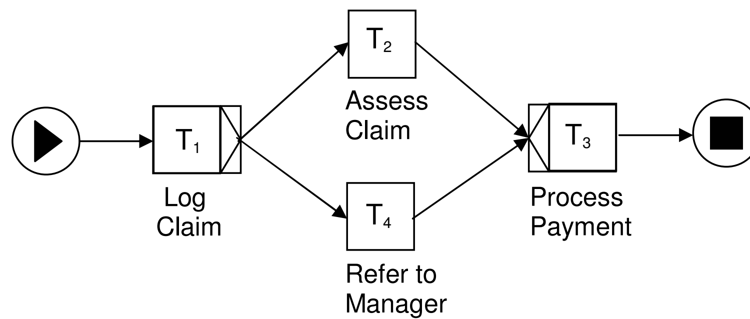

Figure 4.3: A 'view' of Figure 4.2 extrapolated from the modified RDR for task T_2

ditional branches within a standard monolithic workflow schema; of course, a translated rule set of a more than trivial size would demonstrate the complexities of describing the entire set of possible branches monolithically. This approach enables the model to be displayed as the derived view in Figure 4.3, or as the original representation with separate associated worklets, thereby offering layers of granularity depending on factors such as the perspective of the particular stakeholder and the frequency of the occurrence of a condition-set being satisfied. From this it can be seen that an RDR tree may be represented in the modelling notation as a composite set of XOR splits and joins. The advantage of using RDRs is that the correct choice is made dynamically and the available choices grow and refine over time, negating the need to explicitly model the choices and repeatedly update the model (with each iteration increasingly camouflaging the original business logic).

It may also be the case that changes in the way activities are performed are identified, not by an administrator or manager via new business rules, but by a worker who has been allocated a task. Following the example above, after *Log Claim* completes, *Assess Claim* is selected and assigned to a worker's inbox for action. The worker may decide that the generic *Assess Claim* task is not appropriate for this particular case, because this claimant resides in an identified storm-damaged location. Thus, the worker *rejects* the *Assess Claim* worklet via a button on their inbox. On doing so, the system refers the rejection to an administrator who is presented with the set of case data for *Assess Claim* (i.e. its cornerstone case), and the set of current case data.

The administrator then compares the two sets of case data to establish which relevant aspects of the current case differ from *Assess Claim's* cornerstone. Note that while many of the values of the two cases may differ, only those that relate directly to the need to handle this case differently are selected (for example, the location of the claimant and the date the damage occurred). After

identifying the differences, the administrator is presented with a list of possible worklet choices, if available, that may suit this particular context. The administrator may choose an appropriate worklet to invoke in this instance, or, if none suit, define a new worklet for the current instance. In either case, the identified differences form the conditional part of a new rule, which is added to the RDR tree for this task using the rule addition algorithm described earlier.

The principles derived from Activity Theory state that all work activities are mediated by rules, tools and division of labour. Translating that to an organisational work environment, rules refer to business rules, policies and practices; tools to resources and their limitations (physical, financial, staff training, experience and capabilities, and so on); and division of labour to organisational hierarchies, structures, roles, lines of supervision, etc. Of course, these constraints apply to the creation of a new worklet, just as they would in any workflow management system. This means that the authority to create new worklets and add rules to rule sets would rest with the appropriate people in an organisation, and the authority to reject inappropriate worklets would reside within the duties of a worker charged with performing the task — the 'sub-domain expert'. Of course, spurious rejection of tasks would be managed in the same way as any other instances of misconduct in the workplace.

In all future instantiations of a specification, the new worklet defined following a rejection would be chosen for that task if the same contextual conditions occur in a new instance's case data. Over time, the RDR rule tree for the task grows towards specificality as refinements are added (refer Figure 4.1).

4.4 Exception Handling

There are several different types of exception that may occur during the life of a case instance. At various points, constraints may be checked to determine if an exception has occurred. Also, certain tasks may have associated deadlines that have passed, a task being performed by a computerised application or process may fail, or an exception may be raised externally. With all of these events, the context of the current instance is important in determining the appropriate action to take. That action may include suspending, cancelling, restarting or completing a task and/or case instance, as well as performing some compensatory activities. Often a series of such actions will be needed to successfully handle an exception. Also, action may need to be taken at the task level, the case instance level, or may even affect all of the current case instances for

a specification.

In the worklet service, all of the necessary actions are captured in an exception-handling process, or *exlet*. An exlet may contain a series of steps incorporating the actions described above, and may include one or more worklets to perform compensatory activities as part of an exception-handling process.

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 [152]. In the worklet service, each type may have an associated RDR rule tree in the rule set for a specification — that is, a RDR rule set for a specification may contain up to eleven rule trees, one for each exception type and one for selections. So, depending on the exception type that has occurred and the context of the case instance, an appropriate exlet can be chosen and executed to handle the exception.

If there are no rules defined for a certain exception type in the rule set 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.

An invoked exlet may suspend its parent case instance when it is activated, but there may be some occasions when the exlet can (or needs to) operate in parallel to its parent. For example, it is undesirable to suspend an entire *Conference Proceedings* process when a request to withdraw a single paper has been received – it is far easier to handle the request as a (local) externally raised exception while allowing the parent process to continue. Exlets may be defined to accommodate whatever action is desired to be taken.

4.4.1 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 or executed);

- *ItemPostConstraint* - item-level post-constraint rules are checked when each workitem reaches a completed status; and
- *CasePostConstraint* - case-level post constraint rules are checked when a case completes.

The service receives notification from the workflow engine when each of these constraint events occurs within each case instance, then checks the rule set associated with the specification to determine, firstly, if there are any rules of that exception type defined for the specification, 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 associated with a Time Service and the deadline set for that workitem is reached. In this case, the workflow engine notifies the 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, separate timeout rules may be defined (or indeed not defined) for each of the workitems affected by the timeout, including the actual timed out workitem itself. Thus, separate actions may be taken for each affected workitem individually.

Externally Triggered Types Externally triggered exceptions occur, not through the case's data parameters, but because of the occurrence of an event, 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).

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).

4.4.2 Exception Handling Primitives

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

The set of primitives that may be used to construct an exlet 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 task of which the workitem is an instance 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 task of which the workitem is an instance 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 task of which the workitem is an instance 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*: run one or more compensatory processes (i.e. worklets). Depending on previous primitives, the worklets may execute simultaneously to the parent case, or execute while the parent is suspended.

Optionally, an *array* of worklets may be defined for a particular compensation primitive — when multiple worklets are defined for a particular compensation primitive via the Rules Editor, they are launched concurrently as a composite compensatory action when the exlet is executed.

As mentioned previously with regards to the selection service, worklets can in turn invoke child worklets to any depth — this also applies for worklets that are executed as compensatory processes within exlets. The primitives 'Suspend All Cases', 'Continue All Cases' and 'Remove All Cases' may be flagged when being added to an exlet definition via an option 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 (see Section 6.6.3 for an example). Also, the 'Continue' primitives are applied only to those corresponding 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.

In the same manner as the selection service, the exception 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 6.4, the centre tier shows the exlets defined for ItemPreConstraint violations. As mentioned above, there may actually be up to eight 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 for a particular case instance.

The Rollback Primitive One further primitive identified by Russell et al. is 'Rollback' [152], 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 the database processing domain, where it describes 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 it was considered that a rollback action serves no real purpose at the control-flow level (that could not be achieved using the other primitives) 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 counter-productive in an organisational sense. Better to execute a compensation exlet that corrects the problem so that both the original *and* corrective actions are maintained and recorded — 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 readily available. There is enough flexibility inherit in the primitives above to accommodate any kind of compensatory action. As an example, if a loan is approved before it becomes evident that an error of judgement has been made by the approving officer, it is better to run some compensation to redo the approval process (so

that a record of both approval processes remains), rather than rollback the approval process, and thus lose the details of the original approval.

4.5 Service Interface

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 (i.e. deployed on a web server in a location remote to the workflow enactment engine) 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 interface requires a number of events and methods, some originating from the service side and others from the engine side. Some require a response, while others do not require any acknowledgement (such as event notifications that do not necessarily require action).

This section describes the interface requirements, grouped into those required for the selection process and those for the exception handling process.

4.5.1 Selection Interface

The event notifications that must be provided to the service by an enactment engine are:

- a *workitem is enabled* event, where the engine notifies the service that a workitem is ready to be executed. The worklet service will use the event to query the rule set to discover if the enabled workitem has a worklet repertoire and, if so, if one is appropriate to act as a substitute for the workitem. If an appropriate worklet is found, this event becomes the catalyst for a service selection procedure. If an appropriate worklet is not found, the event is simply ignored, allowing the workitem to be executed in the default manner for

the enactment engine. Since it is only necessary for an engine to notify the service if there *may* be available worklets for a workitem, a service-aware enactment engine would allow tasks to be flagged as 'service-enabled' and thus would operate more efficiently than an engine that sent events for *every* enabled workitem, although this is not a necessary requirement of the enactment engine.

- a *work item is cancelled* event, which is necessary to accommodate the situation where the service has substituted a worklet for a workitem and the workitem is cancelled (for example, if it is a member of a cancellation region of the parent process or if the parent process is cancelled). In such cases, the service will need to take appropriate action. Again, a service-aware engine would only generate this event if the service has already substituted the workitem with a worklet, but it it's not a necessary requirement of the enactment engine.

- a *process instance has completed* event, which is required by the service so that it is aware when a worklet instance completes, so that it can finalise the substitution procedure and allow the parent process to continue to the next task.

Each of the events above do not require acknowledgements from the service back to the originating engine — they are simply notifications from the engine that may or may not be acted on.

In addition to the three events, a small number of interface methods are required to be made available to the interface to enable the service to communicate with the engine and take the appropriate action. The required methods are generic in their nature and so would typically be available in most workflow enactment engines. They are:

- *connect to engine*: a connection would typically be required through the interface to enable messaging to pass between the engine and the worklet service.

- *load specification*: the service requires the ability to load a worklet specification into the enactment engine. Since a worklet may be represented as a 'normal' specification of the host enactment engine, this method would already be available within the engine.

- *launch case*: the service must have the ability to launch an instance of a loaded worklet specification.

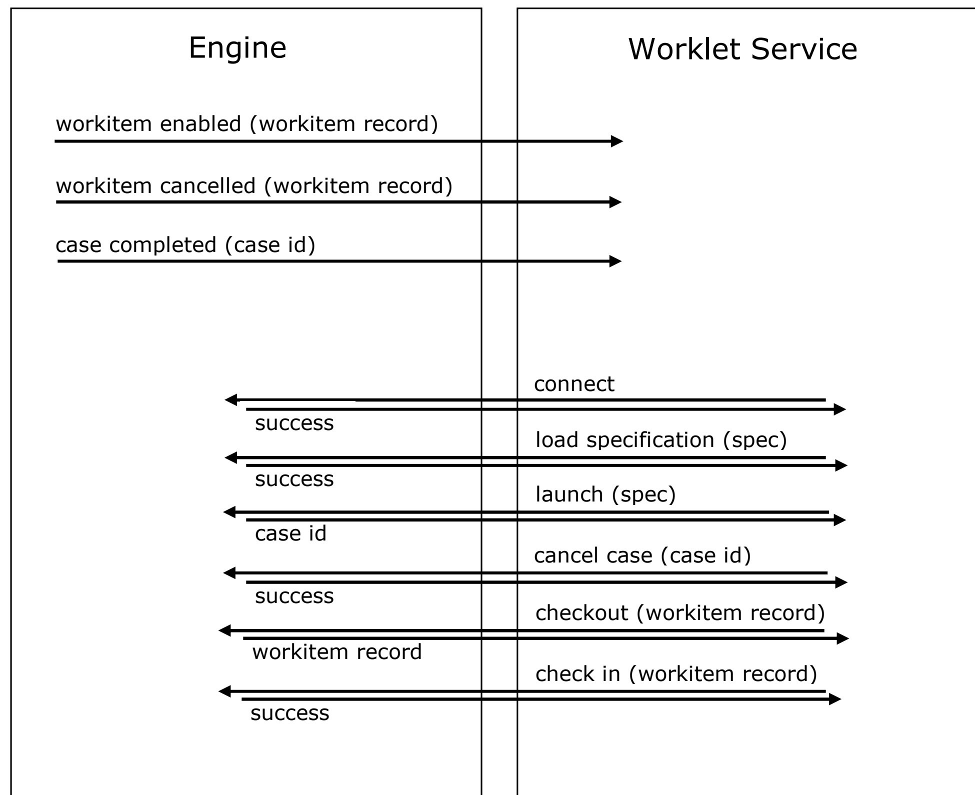

Figure 4.4: Required Selection Interface

- *cancel case*: the service requires the means to cancel a launched worklet instance (for example, if the workitem it has substituted for is cancelled, then the worklet instance would typically need to also be cancelled).

- *check-out workitem*: when the service has been notified of an enabled workitem, and there is a worklet to act as a substitute for it, the service needs a way to take control of the workitem's execution. Thus, by 'checking out' the workitem from the engine, the engine would pass responsibility for the execution of the workitem to the service, and wait for the service to notify it that the execution has completed (i.e. when the worklet case completes).

- *check-in workitem*: when a worklet instance has completed, the service will notify the engine that execution of the original workitem it acted as a substitute for has completed. The data gathered by the worklet would be mapped to the workitem before it was checked in.

Figure 4.4 summarises the interface required by the service's selection process.

4.5.2 Exception Interface

The first four methods described for the selection interface are also used by the exception handling process. In addition, the exception interface requires a further set of events and methods. The enactment engine must notify the service for the following events:

- *check constraints events*, which would notify the service at the launch and completion of each process instance, and at the enabling and completion of each workitem — thus, four unique constraint type events are required. The service would use these events to check that constraint rules have been violated, using available contextual data, and if so, launch the corresponding exlet for that constraint type. Therefore, the constraint event types do not notify of exception *per se*, but are checkpoints that allow the service to determine if a constraint violation has occurred.

- a *time-out event*, which would notify the service when a deadline has been reached, allowing the service to take the appropriate action, if necessary.

- *unavailable resource*, *item abort* and *constraint violation during execution* events. For these event types, the enactment engine must determine if and when these exceptions occur, as opposed to those above where the determination rests with the worklet service.

- *case cancelled event*, required so that the service can take the appropriate action if a worklet instance, or a parent instance for which a worklet is currently executing as a compensation, is cancelled. Note that a case cancelled event is different from a case completed event as described in the previous section.

The following methods are required to be available to the interface for the exception handling service:

- *suspend workitem*: to pause or prevent the execution of a workitem, until such time as it is continued (unsuspended). Through this method, entire process instances may also be suspended, by suspending each 'active' workitem of the process instance as a list.

- *continue workitem*: to unsuspend a previously suspended workitem, or unsuspend each suspended workitem for a previously suspended process instance.

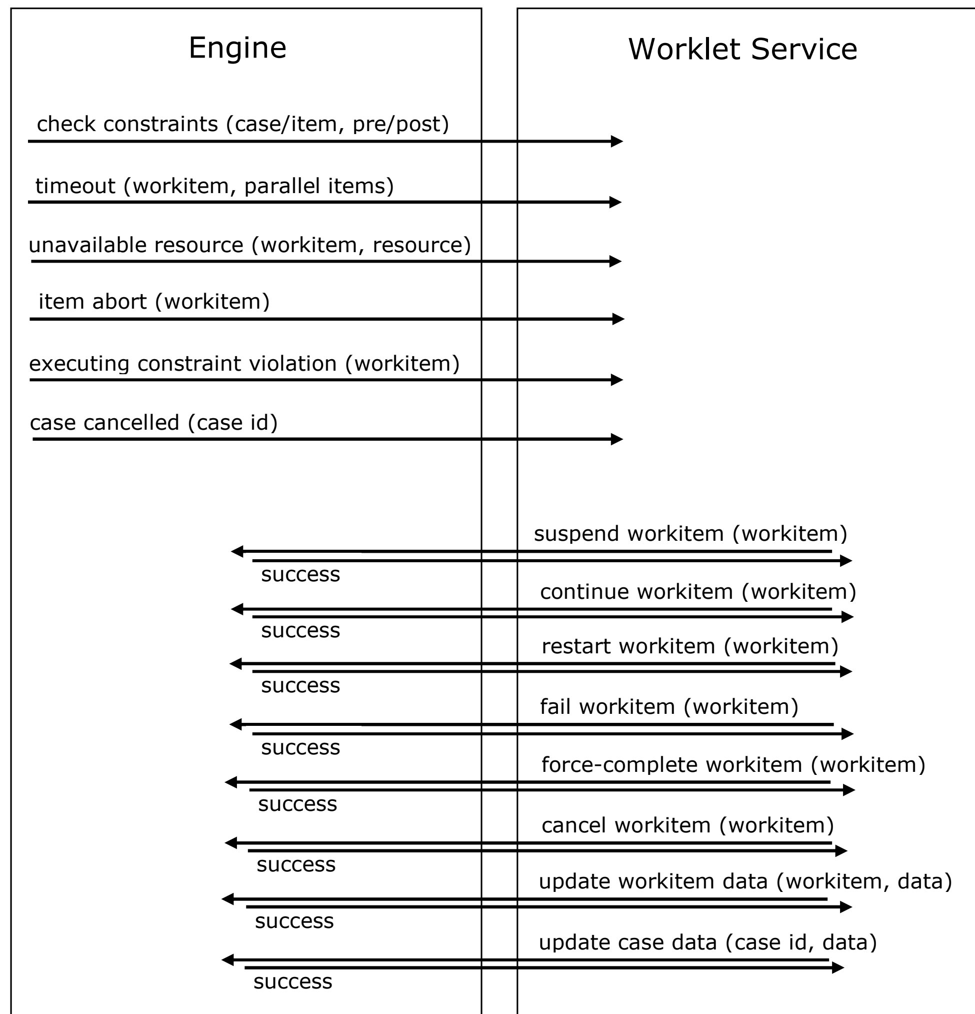

Figure 4.5: Required Exception Interface

- *update workitem data*: for those situations where a worklet has been run as a compensation process for a workitem that has generated an exception, this method would enable any data gathered during the compensation execution to be passed back to the original workitem.

- *update case data*: as above, but for case-level exceptions as opposed to workitem-level exceptions.

- *restart, cancel, force-complete, fail workitem*: to perform an action as specified in an exlet definition.

Figure 4.5 shows the interface requirements for the exception handling procedures of the worklet service.

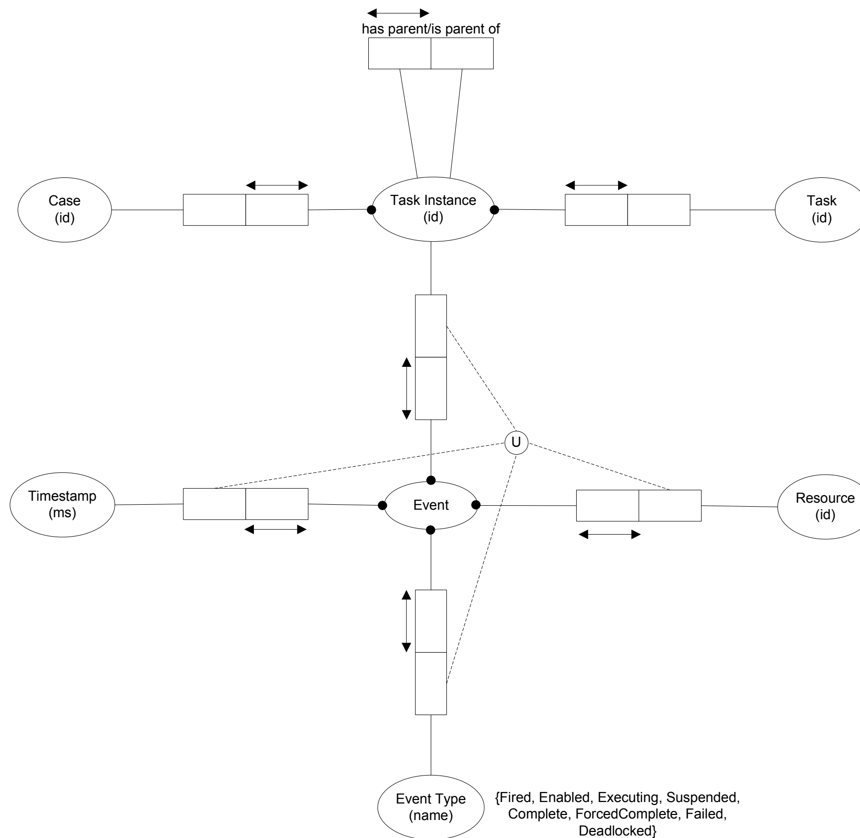

Figure 4.6: ORM Diagram of a Typical Engine Process Log

In summary, it can be seen that the worklet service can successfully manage the flexibility and exception handling needs of a workflow enactment engine via the provision of an interface with a small set of required events and methods. Further, the event types and methods are either of the kind that would already be available in the external interfaces of a number of enactment engines, or represent methods and events that exist internally in enactment engines, and thus may be provided to an interface with a minimal extension to the engine required.

4.6 Secondary Data Sources

When making a contextual choice of an appropriate worklet or exlet, it may be desirable or even necessary to seek data outside the parameters of the task/or case. For example, the current state of the process, or the states of individual tasks, may have some bearing on the choices made; available resources may be an important factor and so on. Thus, choosing the most appropriate worklet or exlet for a situation will be achieved by defining rules that use a combination of currently available data attribute values, both case dependent and independent.

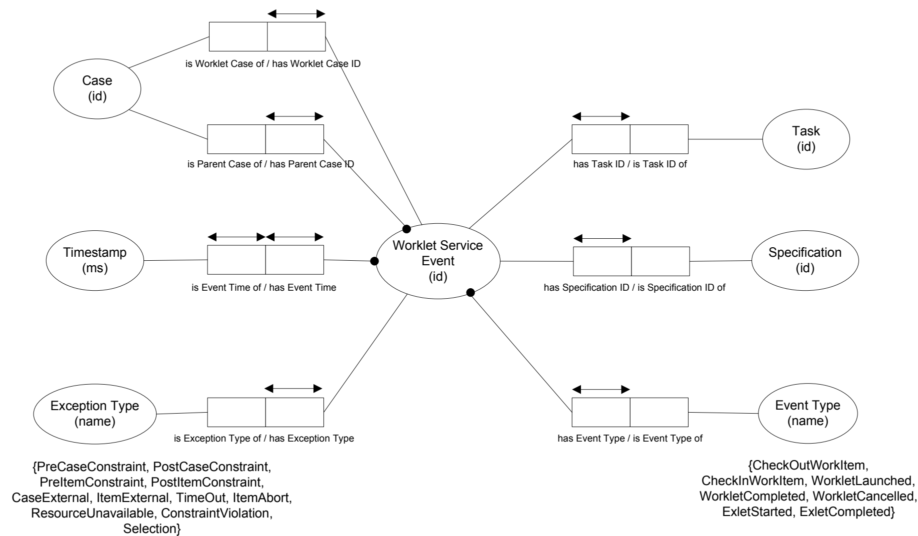

Figure 4.7: ORM Diagram of a Process Log for the Worklet Service

One method of accessing the current set of states for an instantiated process, resource data and archival histories of previous instantiations of a specification may be deduced by mining the process log file [80, 13]. A series of predicates can be constructed to enable the extraction of the current state set and any relations between active worklets and tasks, as well as archival trends and relations. These predicates may then be used to augment the conditionals in RDR nodes to enable selection of the appropriate exception handling exlet.

The kinds of information that may be extracted from the process log file using these predicates include the current status of a worklet, whether a worklet is a parent or child of another worklet, when a certain state was entered or exited for a particular worklet or task, the resource that triggered a state change, and so on [91]. Section 6.8 provides an explanation of how such predicates may be constructed and used in practice).

Figure 4.6 shows an ORM diagram for a generic event log database for a typical workflow enactment engine, while Figure 4.7 shows an ORM diagram for the log kept by the worklet service. The entity ‘Parent Case’ in the worklet log corresponds to the case identifier of the parent process that a worklet was launched for. Hence, the entity would map to a record of the parent case instance in the engine process logs. The worklet log entities ‘Task’ and ‘Case’ would also map to corresponding records in the engine process logs. Thus, by mapping those entity values, a complete view of the process, inclusive of any actions taken on behalf of it by the worklet service, can be constructed.

It should be noted that the event types for the engine log differ from those of the worklet

log. Since worklets are launched as separate cases in the workflow engine, the engine's log records the progress of the process instance, so that it is not necessary to record those types of events in the worklet service logs — indeed, to do so would be a duplication. Thus, the worklet log concentrates on the event details that are not recorded on the engine side, such as CheckOutWorkItem, ExletStarted and so on.

Conclusion

The worklet approach to workflow flexibility and exception handling presents several key benefits, including:

- A process modeller can describe the standard activities and actions for a workflow process, the worklets for particular tasks, and compensation processes for any exception-handling activities, using the same modelling methodology;
- It allows re-use of existing process and exception handling components. Removing the differentiation between exception handling processes and the 'normal' workflow aids in the development of fault tolerant workflows out of pre-existing building blocks [81];
- Its modularity simplifies the logic and verification of the standard model, since individual worklets are less complex to build and therefore verify than monolithic models;
- It provides for 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 to the model each time a business practice changes or an exception occurs is unnecessary; and
- In the occurrence of an unexpected event, an administrator needs simply to choose an existing worklet or exlet or build a new one for the particular situation, which can be automatically added to the repertoire for future use as necessary, thus avoiding complexities including downtime, model restructuring, versioning problems and so on.

Most importantly, the worklet approach is built on the solid theoretical foundations of Activity Theory, and so fully supports the set of derived principles of organisational work practices,

and the criteria for a workflow support system based on those principles, as described in Chapter 3.

Chapter 5

Formalisation

The *Worklet Service* has been designed to make use of a small set of interface methods for communication and data transfer between itself and a workflow enactment engine. For the worklet selection sub-service, an existing interface of the YAWL environment was used, since it provided the necessary communication framework — thus the worklet service conforms to the requirements of a YAWL Custom Service [12, 4] (see Chapter 6 for more details). Custom YAWL services interact with the YAWL engine through XML/HTTP messages via certain interface endpoints, some located on the YAWL engine side and others on the service side. No interface existed within YAWL that provided the framework necessary for the design and operation of the worklet exception sub-service — therefore, that interface was created for YAWL as a product of this research.

The design of the worklet service has been formalised through the definition of a series of *Coloured Petri Nets* (CPN). The main benefit in choosing CPN for the formalisation is that it is a graphical language with formal semantics and a plethora of analysis tools and methods, including the ability to 'execute' or simulate the net to validate the design model [97]. CPN offers a relatively small number of primitives that can be used to build an explicit description of both the potential states of a system and the actions that the system performs. CPN also provides for the analysis of true concurrency [97], which is particularly important for this research, where several actions may take place simultaneously and/or asynchronously, and thus must be handled appropriately. In addition, CP-Nets integrate control, synchronisation and data descriptive layers, making CPN an ideal candidate for the formalisation of a process-aware system such as the worklet service.

CP-Nets have been used to formalise a number of diverse systems in the literature, for example: expressing architectural requirements and assessing middleware support in terms of decoupling [23]; formalising the implementation of a patient record system by modelling task descriptions [102]; modelling a dynamic on-demand routing protocol [174]; simulating business processes [96]; and the modelling of cooperative arrival planning in air traffic control [133].

A Coloured Petri Net is a high-level Petri net in which each token is assigned an (arbitrarily complex) data value, or token *colour*. To populate a given place, a token's colour must conform to the place's data type, called the *colour set* of the place.

The nets in this section have been developed using the package CPNTools, within which colour sets, token colours and their functions, operations and expressions are defined using CPN-ML, an extension of the Standard ML language. Together, the nets form a hierarchical specification of the worklet service. Thus, this chapter describes each net with a view to articulate a technical operational overview of the service.

Formalising the service offers several other benefits. It promotes the understandability of systems and delivers specifications that are unambiguous and able to be analysed [77]. Formal models, such as those created here, are responsive to mathematical manipulation and reasoning, and allow for rigorous testing. They are simultaneously a specification of a system to be built and a presentation to explain the system to others [97]. By formalising the design of a system and then analysing it, the designer gains a much deeper understanding of the proposed system. Finally, the formal model can be expressed as a hierarchy of sub-models, which conforms rather well to the modular construction of a software artefact.

For reference, a complete list of the declarations set (colour sets, variables, constants and functions) for the nets can be found in Appendix A.

5.1 Flexibility: The Selection Service

The design of the Worklet Dynamic Selection Service uses YAWL's Interface B to receive notifications from the engine. Its operations are defined in Figure 5.1.

The interaction of the selection service with a YAWL process consists of three engine events, each represented by an 'In' place on the net, and each event triggers a specific part of the net. The interaction begins when the service receives notification that a workitem has become

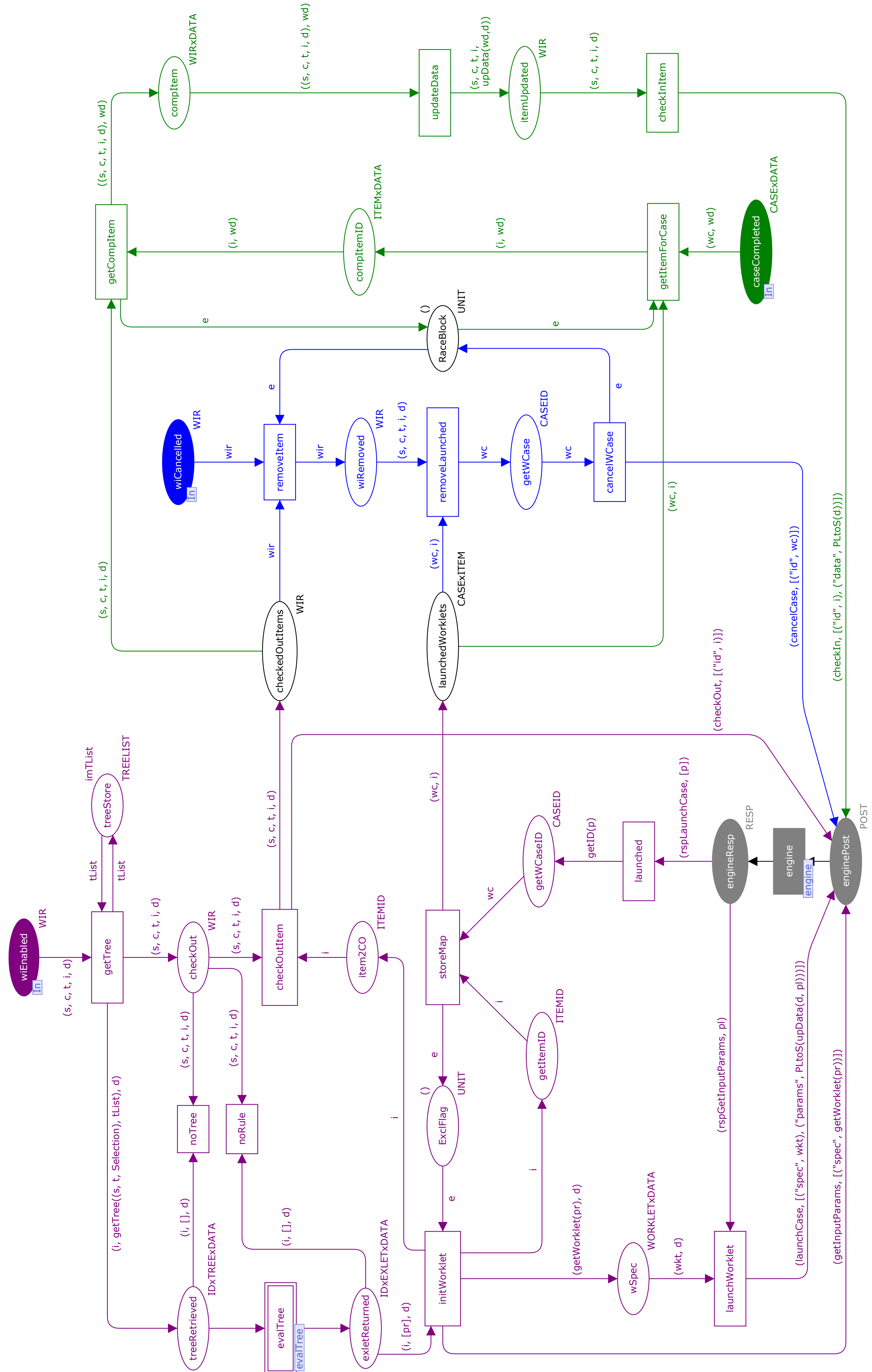

Figure 5.1: CPN: The Selection Service

enabled. This is signified by a token, containing identifiers for the specification, case, task and item, and its list of data parameters, arriving in the *wiEnabled* place. The service will first attempt to retrieve a rule tree from the *treelist* place, which contains a list of all rule trees available to the service; each tree is uniquely identified by a combination of its specification id, task id and exception type. In the case of the selection service, the required 'exception' type is the pseudo-type *Selection*.

If a matching tree is not found, then the workitem's token is consumed by the sink transition *noTree* (the token produced in the place *checkOut* is also consumed at this point); no further processing occurs for the workitem within the service. As a consequence, the enabled task is treated as an 'ordinary' task by the YAWL engine and allowed to execute in the usual manner.

Conversely, if a matching tree *is* found, then it is evaluated with reference to the workitem's contextual data, to derive a conclusion to the appropriate condition (the *evalTree* sub-net is described in detail in Section 5.3). The conclusion forms a textually stored representation of an exlet, which consists of a set of 'primitives', each of which is a pair of the form [action, target]. In the case of a selection rule tree, the conclusion of each rule is an exlet consisting of a single primitive; the value of the target attribute is the worklet specification identifier. Note that selection conclusions use the exlet construct so that both sub-services, selection and exception, can make use of the same rules framework.

If no rule is satisfied with respect to the workitem's dataset, then the *evalTree* transition produces an empty exlet, in which case the token is consumed by the *noRule* sink transition; again the related token is also consumed from the *checkOut* place at this point and no further processing occurs within the worklet service for this workitem. If a rule is satisfied, then the *initWorklet* transition is enabled and performs four actions:

- a token is produced in the *item2CO* place, enabling the *checkOutItem* transition, which fires to call the engine's *checkOut* method and stores the checked-out workitem record in the *checkedOutItems* place;
- an *itemid* token is produced in the *getItemID* place;
- the worklet specification identifier is passed to the *getInputParams* method to retrieve the net-level data parameters for the pending worklet case; and
- the worklet specification id and the workitem's data parameters are stored in the *wSpec*

place.

Several of the nets in this chapter describe a method being passed through the interface to the engine and a response received. The method calls are structured as a product of the name of the method and a set of parameters, each an attribute-value pair. This structure closely mirrors actual interface calls, where method arguments are converted to string pairs at the interface, passed via HTTP *Post* and *Get* messages, and reconstructed on the other side. Expressions on the output arcs of the *engineResp* place filter the enabling of transitions based on which method is being responded to.

Once the net-level parameters are returned from the engine call, the *launchWorklet* transition is enabled, which calls the *launchCase* method to start the execution of the worklet case.[1] The method receives, as arguments, the worklet specification id and a data parameter list, which is a mapping of corresponding values from the workitem's data to the net-level inputs of the worklet (via the *upData* function). The engine's response from the *launchCase* call includes the case identifier for the started worklet case. That case identifier gets mapped to the original workitem identifier token in the *getItemID* place and the product is stored in the *launchedWorklets* place.

The place *ExclFlag* is used to ensure that the response received from the engine for the *getInputParams* and *launchCase* calls correspond with the *itemid* token stored in *getItemID* — in other words, it ensures exclusive use of the interface by a workitem between the firing of the *initWorklet* and *storeMap* transitions.

The first stage of the selection process is now complete: a workitem has been checked out of the engine (signified by a token in the *checkedOutItems* place) and has been substituted with a worklet case (signified by a token in the *launchedWorklets* place). From this point, either of two things can occur: the checked out workitem may be cancelled or the worklet case may complete.

The engine produces a token in the *wiCancelled* place when a previously checked out workitem is cancelled. This may occur when a case, of which the workitem is a member, is cancelled, or if the workitem is a member of a cancellation region within its parent case, or it is cancelled as a consequence of an exlet's response to an exception. For the selection service, the processing of a workitem cancellation is relatively incidental. First, the cancelled workitem's token is removed from the *checkedOutItems* place, then the corresponding token is removed from

[1] The worklet is launched as a separate case, independent of the parent case instance (from the engine's perspective) — the worklet service maintains the relationships between worklets, parent cases and their data sets.

the *launchedWorklets* place — this token contains the case identifier for the worklet executing as a substitute for the workitem. That case identifier forms the argument for the *cancelCase* method, which is called via the interface to the engine, and as a result the worklet case launched as a substitution for the cancelled workitem is itself cancelled. The service has now finalised the processing of that workitem.

The *caseCompleted* place receives a token from the engine when a previously launched worklet case completes; along with the case identifier for the worklet case, the token also contains the output data set of the worklet case. That token enables the *getItemForCase* transition, which extracts the previously mapped workitem token from the *launchedWorklets* place (bound on the worklet case identifier), and passes the workitem identifier with the worklet case data through the *compItemID* place to the *getCompItem* transition. That transition retrieves the original workitem record from the *checkedOutItems* place, and maps it (via a binding on item identifier) to the worklet case data. Finally, the item's data is updated with the results of the worklet case data (by the function *upData*) and checked back into the engine via the *checkIn* method call. The service has now finalised the processing of that workitem.

Given that a launched worklet is executed as a completely separate case in the engine, there is a (remote) possibility that a workitem, for which a worklet has been launched, is cancelled, and before the selection service can act on that cancellation event, the worklet launched for that workitem completes (or vice versa). Because of that possibility, the place *RaceBlock* is included to guard against a race condition and eventual deadlock that may otherwise have occurred when tokens are simultaneously (or almost simultaneously) produced in both the *wiCancelled* and the *caseCompleted* places referring to the same [worklet case, checked out item] pair, since both events rely on exclusive access to the relevant tokens in the *checkedOutItems* and *launchedWorklets* places to successfully complete.

More precisely, without the *RaceBlock* place, there exists the possibility of a token in the *wiCancelled* place causing the extraction of a bound token from the *checkedOutItems* place via the *removeItem* transition, with a simultaneous token in the *caseCompleted* place causing the extraction of a bound token from the *launchedWorklets* place via the *getItemForCase* transition, resulting in no further progress being possible on either path. With the *RaceBlock* place available, the first transition to fire (out of *removeItem* and *getItemForCase*) receives exclusive use of the required bound tokens in both the *checkedOutItems* and *launchedWorklets* places, allowing that first-fired path to successfully complete. While the remote possibility remains for tokens

to arrive at the input place other than that with exclusive use granted via the *RaceBlock* place, those tokens are permanently prevented from entering the net.

This outcome is acceptable because the result of each event is mutually exclusive. That is, the result of a checked out workitem being cancelled is the cancellation of the worklet launched for it, which precludes the possibility of the worklet successfully completing and therefore producing a token in the *caseCompleted* place. Conversely, the result of a worklet case completing is the checking in of the workitem it was launched for. If the workitem has been cancelled in the meantime, the check-in call is simply ignored by the engine. In either outcome, the selection process has successfully completed from a selection service perspective.

5.2 Exception Handling: The Exception Service

The design of the Worklet Dynamic Exception Service uses YAWL's Interface X (created as an output of this research) to receive notifications from the engine. Its operations are defined in Figure 5.2.

Again, this net consists of three distinct parts. The interaction of the exception service with the engine begins when a token arrives at one of the 'In' places to the left of the net (the *external* place is unique in that it does not get triggered by the engine — it receives a token directly from a user's worklist). Each 'In' place will be described in turn.

Whenever the execution of a case begins or ends, a token is placed by the engine in the *caseConstraint* place. This token contains descriptors for the specification, case and task identifiers, the case's data parameters and a boolean which denotes whether the case is commencing or completing (for a case event, the task identifier has a *null* value). A *caseConstraint* event has two effects on the net: it adds (pre-case) or removes (post-case) a token to/from the place *caseMap*, which keeps track of running cases (for use in subsequent parts of the net); and it creates a token in the place *treeRequired* to begin the detection phase of the process. The *treeRequired* token is a product of the identifier of the entity that triggered the event (for a case constraint, the case identifier), the data parameters and a tree identifer consisting of the product of the specification, task and exception type identifiers.

Generally speaking, most of the 'In' places of the net operate in the same way, in that they use the incoming token to construct a token in *treeRequired* describing the YAWL process that

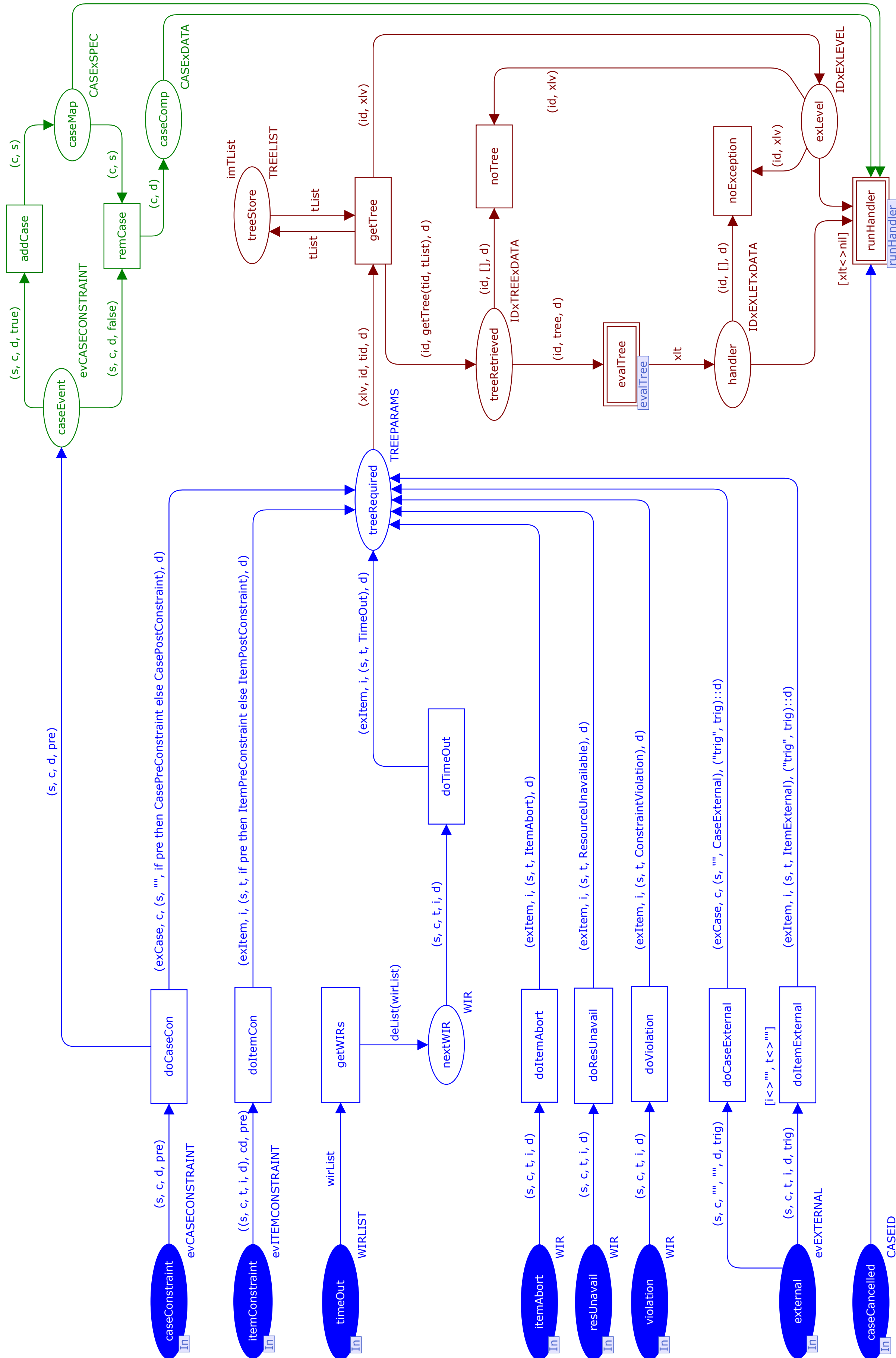

Figure 5.2: CPN: The Exception Service

raised the event, its data and the (potential) exception type.

The *itemConstraint* place receives a token whenever a workitem begins (pre-) or completes (post-). The *timeOut* place's token differs in that it contains a list or workitem records: one for the workitem that reached a deadline, and one for each of the workitems executing in parallel to it; the *deList* function decomposes the list token into a number of workitem tokens, each of which generates a token to the *treeRequired* place — that is, each workitem passed in the list may raise its own exception handler independently. The *itemAbort*, *resUnavail* and *violation* places are unremarkable, while the *caseCancelled* place passes its tokens directly to the *runHandler* sub-net. The *external* place receives process identifiers and a 'trigger' or cause for the raising of the exception, selected from a list displayed to the user. The token it constructs depends on whether it is a case- or item-level token; in either case, the trigger selected is temporarily added to the data parameters so that its value can form part of the conditional expressions to be evaluated in the *evalTree* sub-net.

The detection phase of the net determines if an exception of the type passed has actually occurred (by querying the rule set for the exception against the contextual case data) and, if so, how to handle it. The phase begins by extracting from the *treeStore* place a rule tree that matches the designated specification, task and exception type identifiers (via the function *getTree*), and producing the resultant token in the *treeRetrieved* place. The transition *getTree* also extracts the exception level flag (that is, whether the event occurred at the case or workitem level) and stores it in a token in the *exLevel* place. The exception level is required if a subsequent exception handler invokes a compensation action (see Section 5.8).

If there is no tree defined for the particular identifier combination, then the token is consumed by the *noTree* sink transition (along with the token in the *exLevel* place) and processing of the event completes. Thus, it is not necessary to construct a rule tree for every specification/-task/exception type, but only those for which an exception event is required to be handled (or tested for) — any events that don't have a corresponding rule tree are essentially ignored. If the produced token does contain a tree, it is passed to the *evalTree* sub-net for processing. Similarly, if *evalTree* returns a nil (or empty) exlet, then the token is consumed by the *noException* sink transition (again, along with the token in the *exLevel* place). This signifies that, while a rule tree was found for the token, there was no condition in the rule set that was satisfied. Finally, if an exlet was produced by *evalTree*, then it is passed, along with the the case/task identifier and data parameters, to the *runHandler* sub-net, where the exlet is executed.

5.3 Evaluating the Rule Tree

Figure 5.3 shows the *evalTree* sub-net, used by both the selection and exception services to evaluate the ripple-down rule set associated with the specification/task/exception type identifer product. Thus, this net describes the operation of ripple-down rules within the worklet service and how an RDR rule tree is navigated to return the most appropriate worklet/exlet for a given context.

This net receives from its parent net a token in the 'In' place *treeReceived* consisting of the product of the identifier of the case/workitem that generated the event, the matching rule tree and the associated contextual case/workitem data parameters. For the purposes of the CP-net representation, the tree is structured as a list of rule nodes (although the actual implementation structure will be a linked list). The net firstly splits the token values into three distinct tokens: one which contains the tree, one the identifier and one the data parameters, and stores each in their respective places. The first node in the tree (the root node) is popped from the front of the tree on the incoming arc to the *storeArgs* place, and so will be the node queried first. Note also that an *ExclFlag* place is again used to ensure that only one tree is being interrogated at any one time.

A rule node consists of a node identifer, a conditional expression, its conclusion, a set of 'cornerstone' case data, and pointers to its child nodes (if any). Within the tree, child pointers are represented as the node identifier values of a node's true and false child nodes; a value of (-1) signifies there is no child on that branch.

The *nextNode* place passes the node token to the *evalNode* transition, which uses the data parameters in *dataStore* to evaluate the node's conditional expression to a boolean result, via the *eval* function. If the expression evaluates to true, the *resTrue* transition fires which stores the node token in the *lastTrue* place (replacing the previous token residing there), and produces a node identifier token in the *hasNext* place, being the node identifier of its 'true' child (if any) via the *getChildID* function.[2] If the condition evaluates to false, then the *resFalse* transition fires and the *getChildID* function produces the node identifier of its 'false' child (if any) in the *hasNext* place.

If the token produced in *hasNext* has a value greater than (-1), then the node for that node

[2]The evaluation of a ripple-down rule tree, when complete, returns the last node that evaluates to true; traversal continues while child nodes exist on the relevant branch and completes when there are no more nodes on that branch to evaluate. See Section 4.2 for a detailed explanation of ripple-down rules.

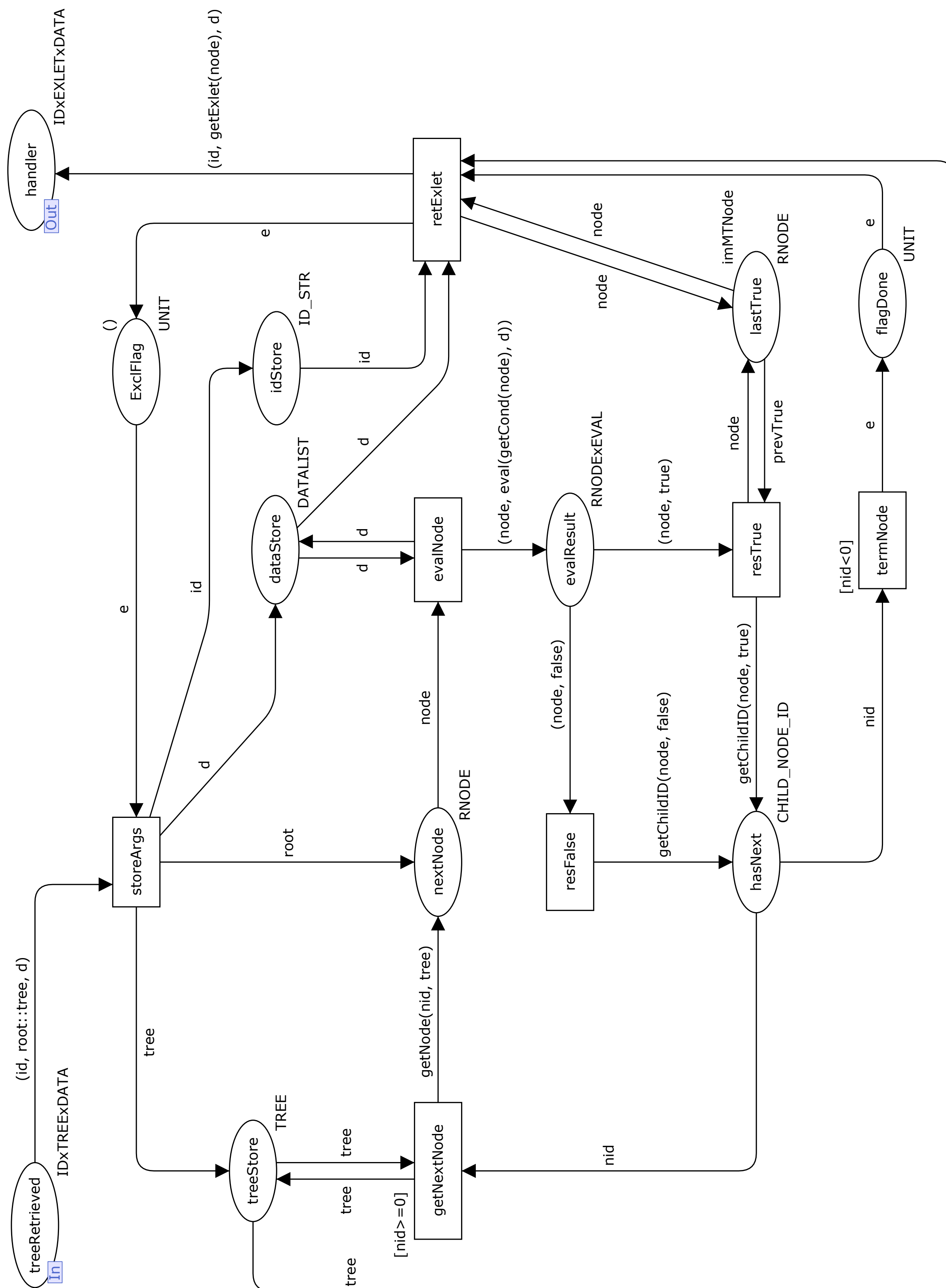

Figure 5.3: CPN: Evaluating the Rule Tree

identifier is extracted from the *treeStore* using the *getNode* function, a token containing the node is produced in the *nextNode* place, and the evaluation process enters another loop. When an evaluated node does not have a corresponding child identifier, the token in *hasNext* will contain a value of (-1), signifying tree traversal has completed, and so the *termNode* transition will fire. This, in turn, causes the *retExlet* transition to enable (through the *flagDone* place). The *retExlet* transition:

- takes a copy of the token in *lastTrue*;
- consumes the tokens in *treeStore*, *dataStore* and *idStore*;
- resets the *ExclFlag* place to allow the next tree evaluation to occur; and
- returns a token via the *handler* place that consists of the product of the original id and data values, and the exlet of the last node in the traversal that evaluated to true (via the *getExlet* function).

At the completion of a net instance, a token remains in the *lastTrue* place representing the last node that has its condition evaluate to true. However, this poses no problem because, on the next iteration of the net, the token will be immediately removed and replaced with a token representing the root node of the new tree presented to the net's 'In' place (since the root node by definition always evaluates to true). Thus, in every instance, the token that is finally produced in the net's 'Out' place contains an exlet extracted from a node that is a member of the tree presented at the 'In' place.

It is important to note that a *Selection* rule tree will return an exlet with a single 'primitive' — containing the worklet to as act a substitute for a worklet-enabled task — while the exlets for the other exception types may contain multiple primitives.

5.4 Executing an Exception Handler

Once it has been established that there is an exlet resulting from a tree evaluation (that is, an exception has actually occurred within a given context), that exlet, which for the exception service describes the exception handling process, will be executed. The operation of an exception handler is described in the *runHandler* sub-net, as shown in Figure 5.4.

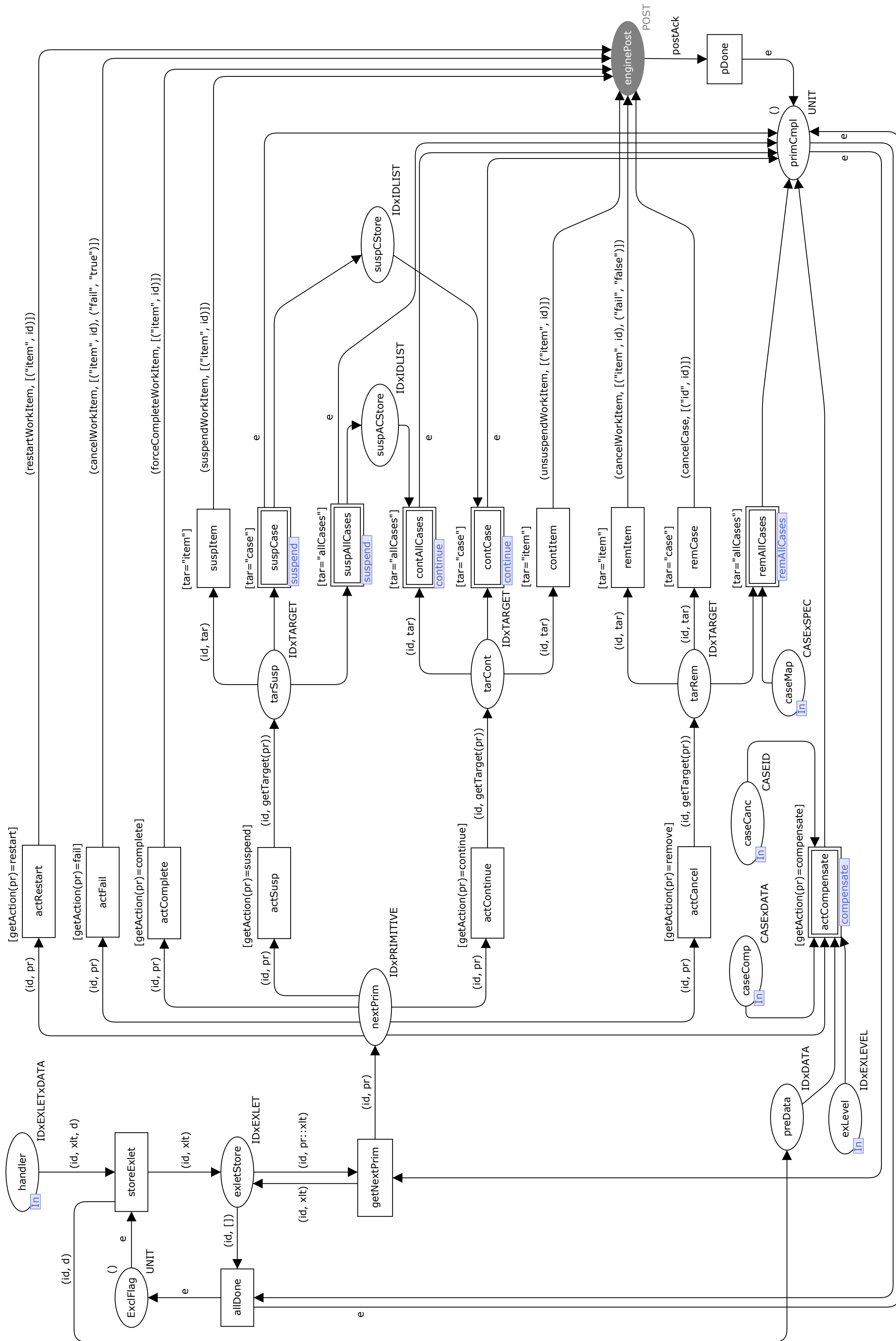

Figure 5.4: CPN: Execution of an Exception Handler

The 'In' place *handler* in the top-left corner receives a token that contains the product of the identifier of the exception's originator (case identifier for case-level or item identifier for item-level exceptions), the exlet to execute and the original data parameters used.[3] A token containing a product of the case/workitem identifier and the exlet is produced in the place *exletStore* for processing. An *ExclFlag* place is used again here, this time to ensure that each exlet gets exclusive use of the net until its execution has completed.[4]

The transition *getNextPrim* pops the next primitive (an [action, target] pair) from the head of the exlet and passes it to the *nextPrim* place, returning the remainder of the exlet back to the *exletStore*. There are seven output arcs emanating from *netPrim*, one for each type of 'action' that a primitive may reference: *restart, fail, complete, suspend, continue, remove* and *compensate*. Thus, each input transition on those arcs has a guard expression to ensure exactly one is enabled by each token produced in the *netPrim* place.

The actions *restart, fail* and *complete* are only relevant when applied to a workitem (as opposed to a case), so each of those requires a call to the appropriate method to be passed to the engine. The actions *suspend, continue* and *remove* may target a workitem, case or all cases (that is, all executing instances of the same specification as the case identifier passed); each of those actions is therefore split three ways and the appropriate transition enabled depending on the target specified in the primitive (as shown in the guard expression of each of those transitions).

For the *suspend* action, the *tarSusp* place has three output arcs. If the target is a workitem, the relevant method call is passed to the engine. When the target is a case or all cases, the suspensions are handled by the *suspend* sub-net (see Section 5.5). While both targets use the same sub-net, the output of each (a list of suspended workitems) is deposited in a separate place: *suspCStore* for case suspensions and *suspACStore* when all cases are suspended. The *continue* action is structured similarly to *suspend* (see Section 5.6).

The workitem list output by a *suspend* action becomes the input to a *continue* action — it is assumed that a *suspend* action has occurred for a particular workitem or case in the current exlet prior to a *continue* action being processed.

A *remove* action will cancel a workitem, case or all cases of a given specification. The action for all cases is handled by the *remAllCases* sub-net (Section 5.7); workitem and case

[3]A token containing the product of the case/workitem identifier and the data parameters is immediately produced to the *preData* place for use by the *actCompensate* sub-net — described in Section 5.8.

[4]In practice, there may be many parallel instances of this net.

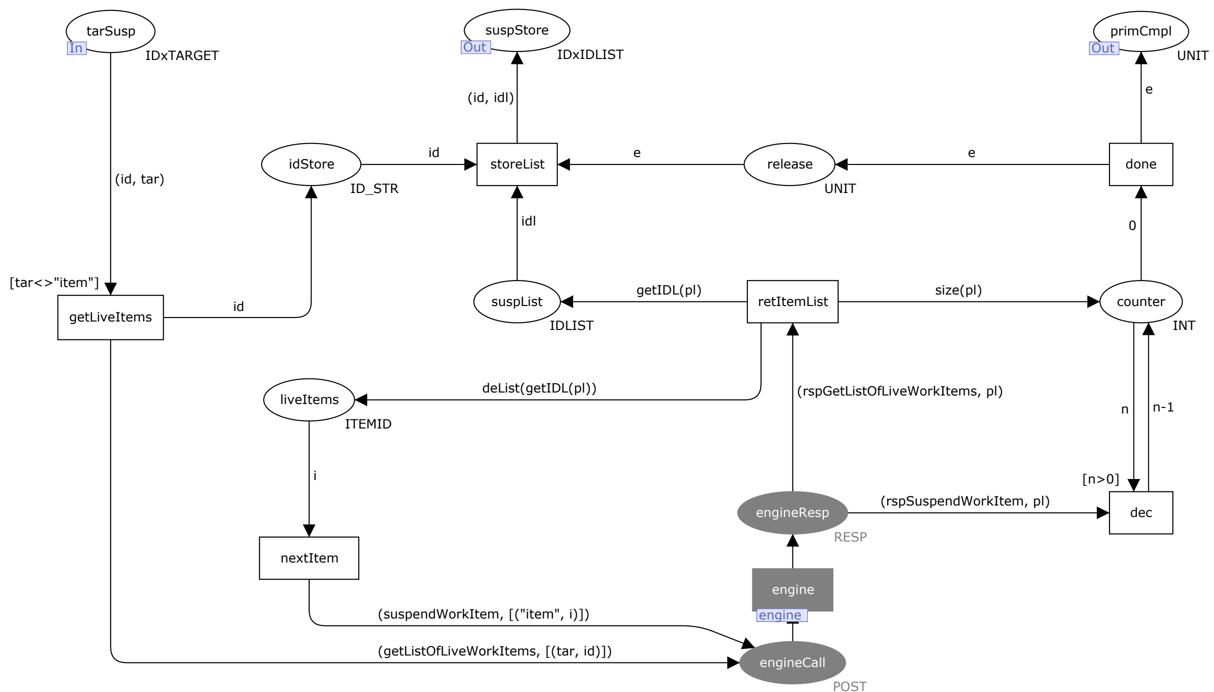

Figure 5.5: CPN: Case Suspension

targets involve the appropriate method call passed to the engine. Finally, a *compensation* action will invoke the *actCompensate* sub-net (Section 5.8).

All actions signal their completion by producing a token in the *primCmpl* place, which in turn re-enables the *getNextPrim* transition to begin processing of the next primitive in the exlet. When processing of the last primitive has completed, the *allDone* sink transition is enabled by tokens in *primCmpl* and *exletStore* (with an empty exlet) to denote that exception handling has completed; the *ExclFlag* is reset for the next handler passed to the net.

5.5 Suspending a Case

The *suspend* sub-net manages the suspension of a case, or of all cases for a specification, depending on the value of the primitive's target. The net is shown in Figure 5.5.

Net execution begins when a token arrives in the 'In' place *tarSusp*, containing a product of the identifier of the item or case that raised the exception and the target — the guard expression on the *getLiveItems* transition ensures this sub-net is not invoked for a workitem target. At that transition, the identifier is stored for later output, while the target forms an argument to the *getListOfLiveWOrkItems* method: if the target is 'case', a list of live work items for the

case is returned, if 'allCases', a list of the live workitems of all current instances of the case's specification is returned. When this method is called, the engine returns the list of items to the *retItemList* transition via a token produced in the *engineResp* place — the response is matched to the call via the output arc expressions of that place.

The exception service will suspend a case at the workitem level, by retrieving a list of all live workitems for that case and then suspending each of those workitems. Transition *retItemList* produces tokens in three places: the list of workitems (to be) suspended is stored in *suspList*; the list is deconstructed into a number of item identifier tokens via the *deList* function and output to *liveItems*; and a count of the number of workitems (to be) suspended is placed in *counter* using the *size* function.

For each item identifier token in the *liveItems* place, the *nextItem* transition fires to call the *suspendWorkItem* method, using the item identifier as an argument. For each response from the engine that an item has been suspended, the value in the *counter* place is decremented. When all live workitems have been suspended, the counter reaches a zero value, which enables the *done* transition, which in turn releases the list of workitems suspended, mapped to the originating workitem or case identifier, to the 'Out' place *suspStore* and produces a token in the *primCmpl* place to denote execution of the primitive has completed. The *suspStore* place may represent one of two places on the parent *runHandler* net, depending on whether a single case (in which case the corresponding place is *suspCStore*) or all cases matching the specification (the *suspACStore* place) have been suspended (cf. Section 5.4).

5.6 Continuing a Case

The *continue* sub-net unsuspends or continues the set of previously suspended workitems; its structure is relatively straightforward (Figure 5.6).

The *contList* transition consumes tokens from the two 'In' places: *tarCont* passes the identifier to bind to a token in *suspStore*, the resultant list of workitem identifiers is deconstructed into a number of individual workitem identifier tokens in the *idlStore* place, and a token containing the number of workitems is produced in the *counter* place using the *size* function. The *unsuspendWorkItem* method is called for each workitem token, and each response from the engine decrements the counter. When all workitems have been unsuspended, the *done* transition

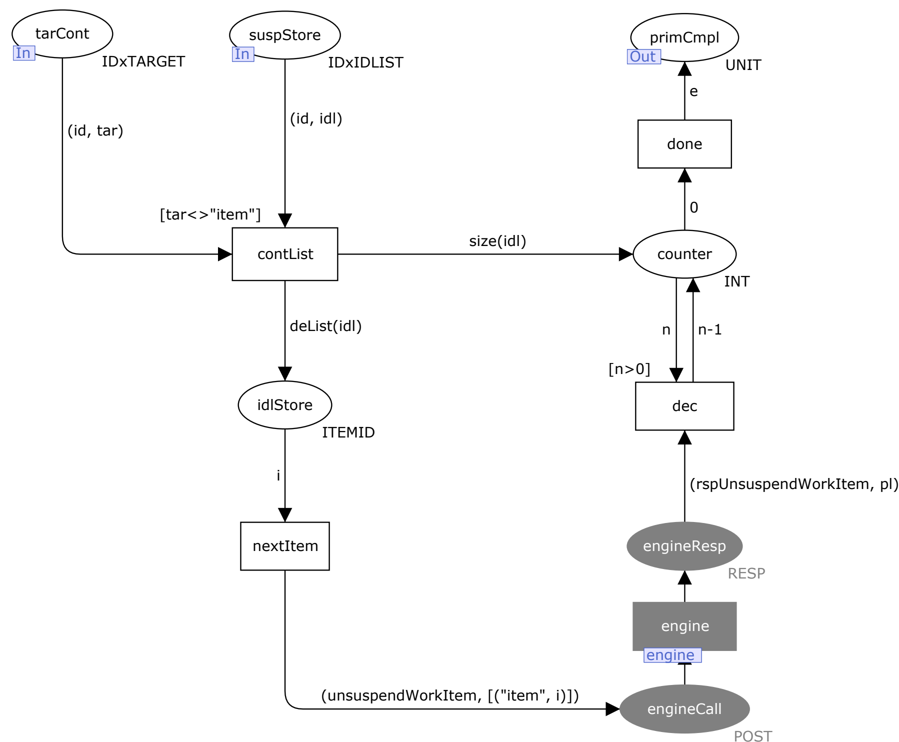

Figure 5.6: CPN: Case Continue

is enabled, producing a token in *primCmpl* to denote execution of the primitive has completed.

5.7 Removing all Cases

The *remAllCases* sub-net cancels all case instances of those cases with the same specification as the case specified (Figure 5.7). Its structure has similarities to the previously described *suspend* and *continue* nets.

The 'In' place *tarRem* receives a token containing the identifier of the case that raised the exception, and binds it to a token which is consumed from the 'In' place *readMap* (this token was originally created in the top-level exception service net at Figure 5.2). That binding allows the specification identifier for the case to be retrieved (via the enabling of the *getSpecID4Case* transition) to become an argument in the *getCases* method call. The engine responds to that method with a list of case identifiers for all the current cases of that specification. That list is deconstructed using the *deList* function into a number of individual case identifier tokens in the *caseList* place; each of those case identifiers is passed in turn to the *cancelCase* method. The 'counter' construct is again used to place a token in the *primCmpl* place to denote completion of the primitive once all of the cases have been cancelled.

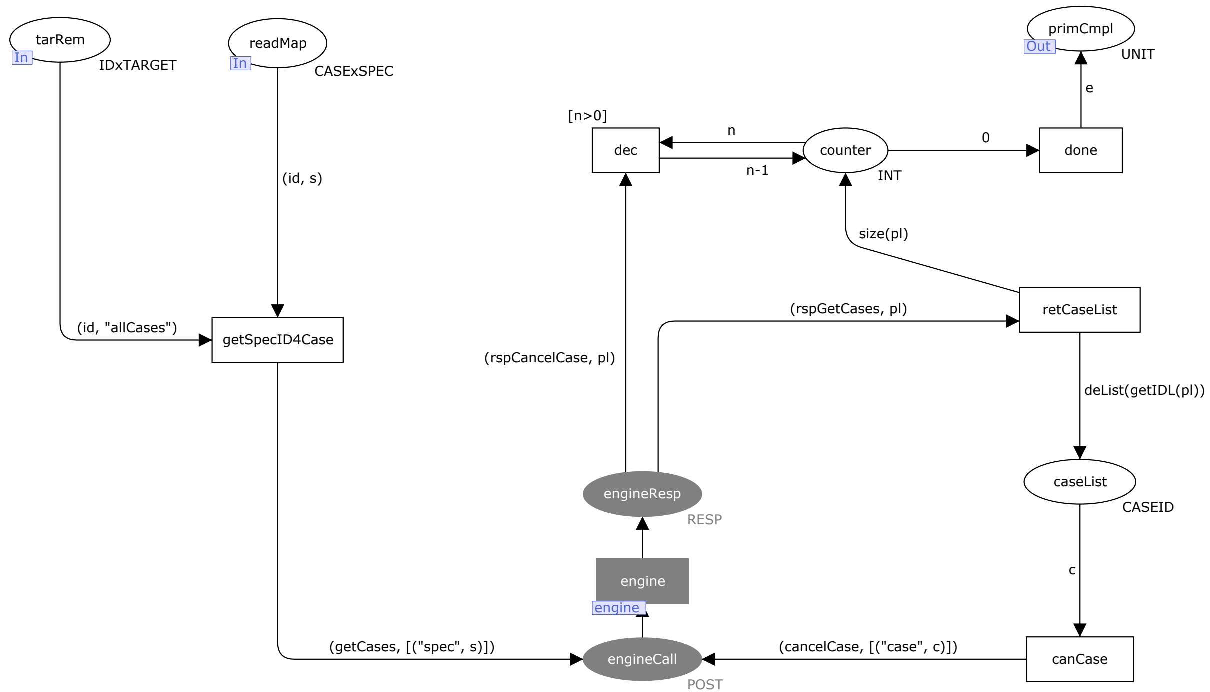

Figure 5.7: CPN: Remove All Cases

5.8 Compensation

The *compensate* sub-net (Figure 5.8) describes how a compensation process (that is, a worklet) is executed by the exception service as part of an exlet. The net has similarities to the related portion of the selection service net (Figure 5.1).

The net has five 'In' places and one 'Out' place. It begins by receiving tokens in the *nextPrim* and *preData* places, and enables the *initWorklet* transition, binding on the item or case identifier. That transition calls the *getInputParams* method to retrieve from the engine the net-level input parameters for the worklet specification (the specification identifier is extracted from the primitive passed using the *getWorklet* function). The returned parameters are updated with the mapped data from the case or item data set passed in to the net, then the worklet is launched via the *launchCase* method. The case identifier for the worklet instance is returned from the engine as a result of the *launchCase* call and mapped to the original case or workitem identifier which becomes the value of the token that is produced in the *launchedWorklets* place.

From there, one of two events may occur. Firstly, a token may arrive in the *caseCancelled* place[5] which will contain the identifier of a case cancelled by the engine — that token will

[5]This place receives its token directly from the engine, originating in the net hierarchy at the exception service net at Figure 5.2.

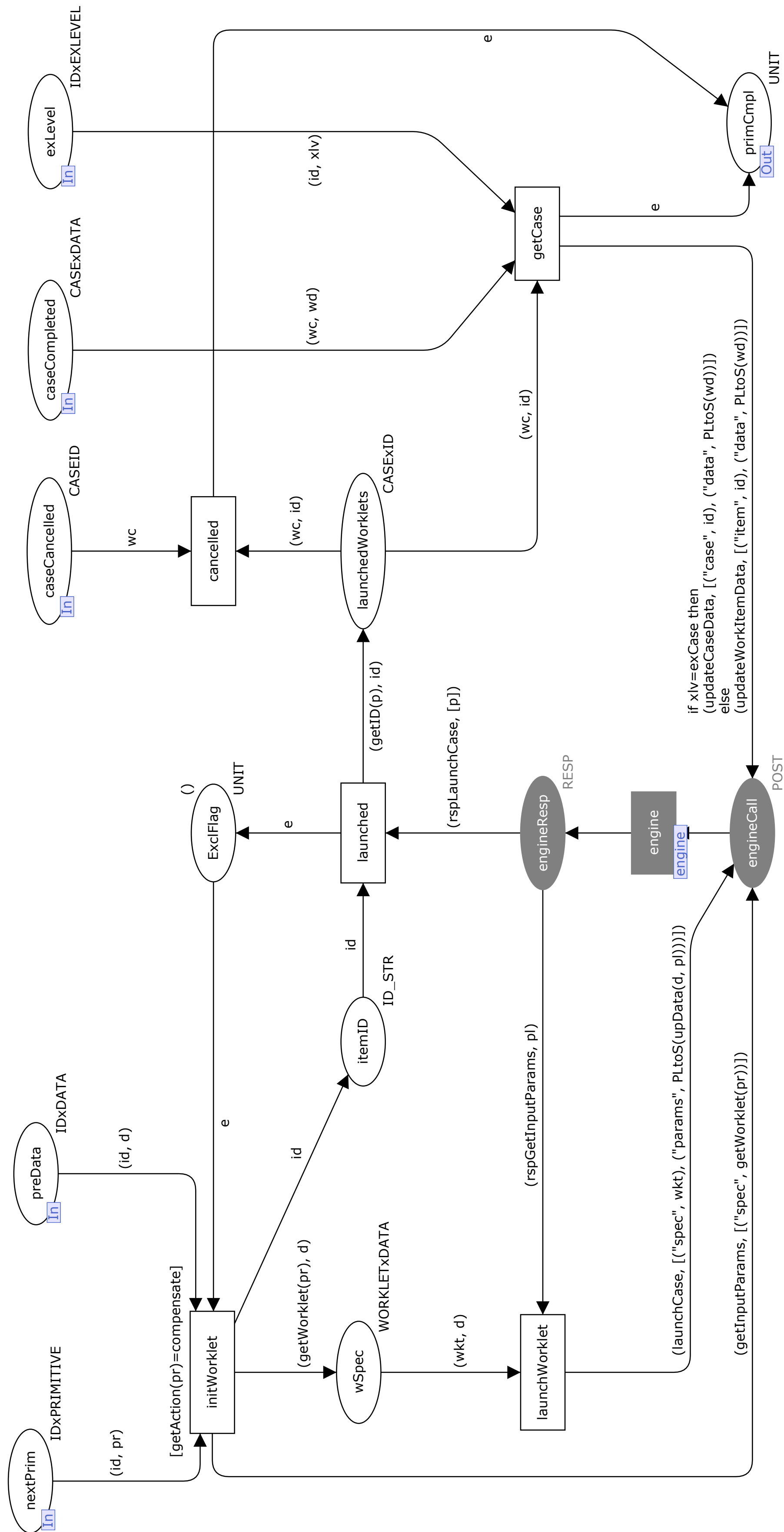

Figure 5.8: CPN: Compensation

bind with and remove a previously stored token in the *launchedWorklets* place. Otherwise a token may arrive at the *caseCompleted* place (following a *PostConstraintEvent*) — that token will contain the completed case and its final output data parameters, which are mapped to the removed token (bound on identifier) from *launchedWorklets*. The *getCase* transition will fire either an *updateCaseData* or *updateWorkItemData* method call, depending on the value of the token from the *exLevel* place (which flags whether the exception was raised at the workitem or case level), to update the data parameters stored in the engine with the mapped values of the worklet's output parameters.

Finally, a token is produced in the *primCmpl* place to denote completion of the compensation process and allow the processing of the next primitive in the exlet (if any). Note that, unlike the other primitives, a compensation primitive may take some time to complete (that is, the time it takes for the worklet to complete or be cancelled).

Chapter 6

Implementation

The worklet service has been implemented as a YAWL Custom Service (see Section 6.3). The selection sub-service was implemented first, in part to ensure proof of concept but also to construct the framework for the overall approach, and was deployed as a member component of the YAWL Beta 7 release. The exception sub-service was then added as an extension to the selection service and was deployed as a component member of the YAWL Beta 8 release.

In the discussion that follows, a reference to the 'worklet service' applies to the entire custom service, while a reference to the 'selection service' or the 'exception service' applies to that particular sub-component of the service.

Figure 6.1 shows the notation of the YAWL language [4]. All of the process models in this dissertation are expressed using this notation — the language is used to model static YAWL processes, and the worklets used both for selection and exception compensations, thus all worklet specifications are examples of standard, complete YAWL process models. A schematic of the external architecture of YAWL, showing the relation of the Worklet Service within it, is shown in Figure 6.3.

The worklet service (including the rules editor), its source code and accompanying documentation, can be freely downloaded from `http://www.sourceforge.net/projects/yawl`.

6.1 Service Overview

The *Worklet Service* comprises two discrete but complementary sub-services: a *Selection Service*, which enables dynamic flexibility for process instances, and an *Exception Service*, which

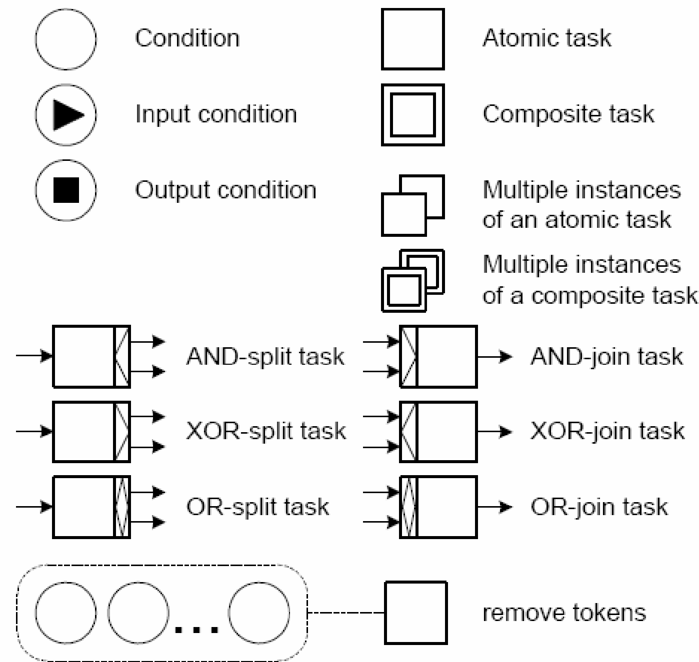

Figure 6.1: YAWL Language Notation [4]

provides facilities to handle both expected and unexpected process exceptions (i.e. events and occurrences that may happen during the life of a parent process instance that are not explicitly modelled) at runtime.

6.1.1 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 is a complete extended workflow net (EWF-net) compliant with Definition 1 of the YAWL semantics [12]. Each worklet instance is dynamically selected and invoked at runtime and may be designed and provided to the Selection Service at any time, *even while a parent process instance is executing*, 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 ripple-down rules to determine the most appropriate substitution.

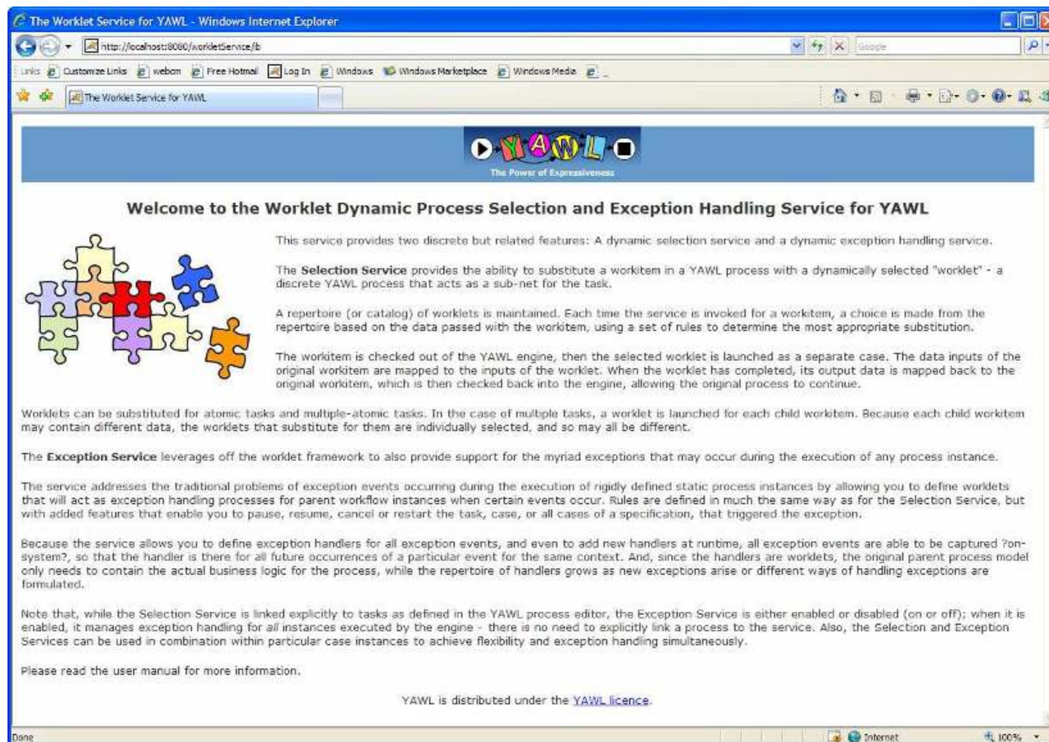

Figure 6.2: The Welcome Page for the Worklet Service

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.

The worklet executed for a task is run as a separate case in the engine, so that, from an engine perspective, the worklet and its parent are two distinct, unrelated cases. The worklet service tracks the relationships, data mappings and synchronisations between cases, and creates 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 instance. See Appendix B for a set of sequence diagrams describing the selection service's processes.

Worklets may be associated with either an atomic task, or a multiple-instance atomic task (cf. Figure 6.1). Any number of worklets can form the repertoire of an individual task, and any number of tasks in a particular specification can be associated with the worklet service. A worklet may be a member of one or more repertoires — that is, it may be re-used for several distinct tasks within and across process specifications. In the case of multiple-instance tasks, a separate worklet is launched for each child workitem. Because each child workitem may

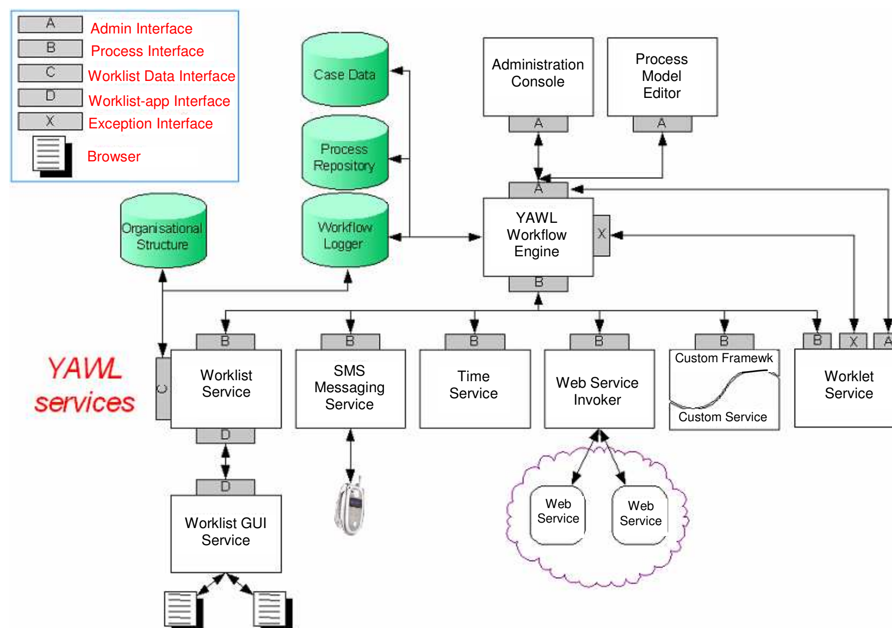

Figure 6.3: YAWL External Architecture (adapted from [4])

contain different data, the worklets that substitute for them are individually selected, and so may all be instances of different worklet specifications.

6.1.2 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 'off-line' 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 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, or of any invocation of handling processes — all exception checking and handling is provided by the service. 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 any need to modify the original process definition.

The exception service uses the same repertoire and dynamic rules approach as the selection service. In fact, the exception service extends from the selection service and so is built on the same framework. 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 — see Section 6.6). 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 6.6).

Table 6.1 summarises the differences between the two sub-services (the interfaces are described in the next section). 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.

An extensible repertoire of exlets is maintained by the service for each type of potential exception within each workflow specification. Each time the service is notified of an exception event, either actual or potential (i.e. a constraint check) the service first determines whether an exception has in fact occurred, and if so, where a rule tree for that exception type has been defined, makes a choice from the repertoire based on the type of exception and the data attributes

Table 6.1: 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

and values associated with the workitem/case, using a set of rules to select the most appropriate exlet to execute (see Sections 4.2 and 6.9).

As for a worklet launched by the selection service, if an exlet executed by the exception service contains a compensation action (i.e. a worklet to be 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. Figure 6.4 shows the relationship between a ‘parent’ process, an exlet repertoire and a compensatory worklet, using an *Organise Concert* process as an example[1]. Since a worklet is launched as a separate case, it is treated as such by the worklet service and so may have its own worklet/exlet repertoire.

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. Like the selection service, the exception handling 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.

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 and requirements for flexibility.

[1] Details of the notations used can be found in Figure 6.1 and Section 6.9.5.

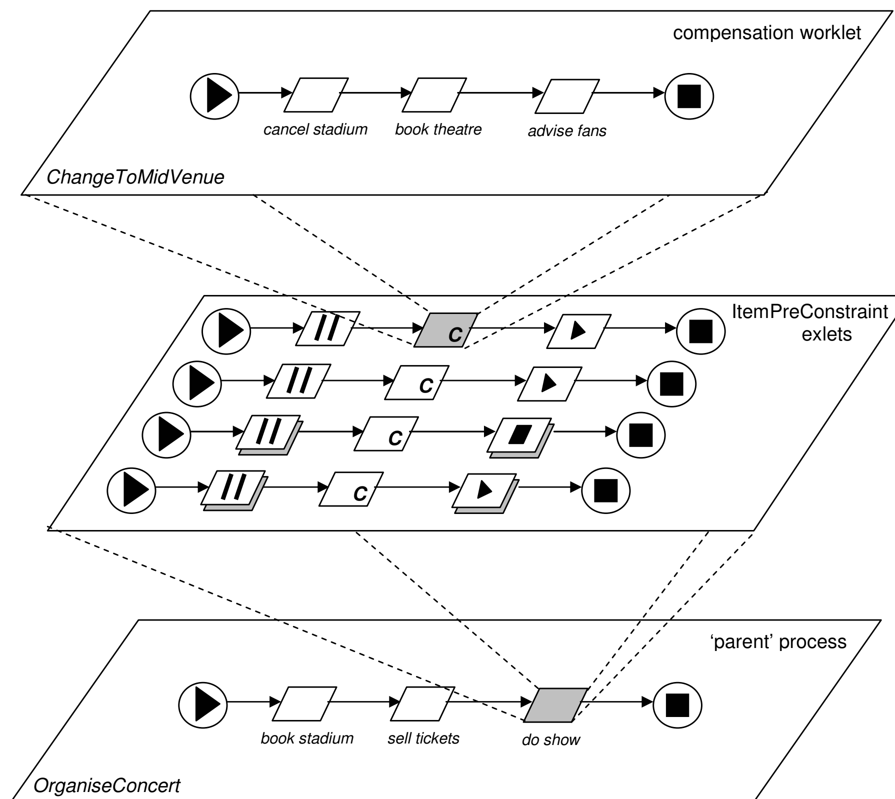

Figure 6.4: Process – Exlet – Worklet Hierarchy

6.2 Service Oriented Approach

The *Worklet Service* has been implemented as a YAWL Custom Service [12, 4]. 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 [9], together with a formal semantics [9]. 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 has a service-oriented architecture, allowing the worklet paradigm to be developed as a service 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 in workflows, orthogonal to the underlying workflow language, is provided.

Custom YAWL services interact with the YAWL engine through XML/HTTP messages via certain interface endpoints, some located on the YAWL engine side and others on the service

side. Specifically, custom services may elect to be notified by the engine when certain events occur in the life-cycle of nominated process instantiations (i.e. when a workitem becomes enabled, when a workitem is cancelled, when a case completes), 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.

For example, on receiving notification from the engine of a workitem-enabled event, a custom service may elect to 'check-out' the workitem from the engine. On doing so, the engine marks the workitem as *executing* and effectively passes operational control for the workitem to the custom service. When the custom service has finished processing the workitem it will check it back in to the engine, at which point the engine will mark the workitem as *completed*, and proceed with the process execution. It is this interaction that is the fundamental enabler of the worklet selection service.

Three interfaces to the YAWL engine are used by the Worklet Service (cf. Figure 6.6):

- **Interface A** provides endpoints for process definition, administration and monitoring [4] — the worklet service uses Interface A to upload worklet specifications to the engine;
- **Interface B** provides endpoints for client and invoked applications and workflow interoperability [4] — used by the worklet 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 worklet service with the necessary triggers to dynamically capture and handle process exceptions.

The logical layout of Interface X can be seen schematically in Figure 6.5, which shows that a custom service (in this case, the exception service) implements the *InterfaceXService* (java) interface, which defines seven methods to be instantiated that enable the handling of notifications that are sent from the *EngineSideClient* object to the *ServiceSideServer* object. The custom service may use the methods of the *ServiceSideClient* object to call complementary methods in the *EngineSideServer* object, thus enabling inter-service communication, using XML over HTTP.

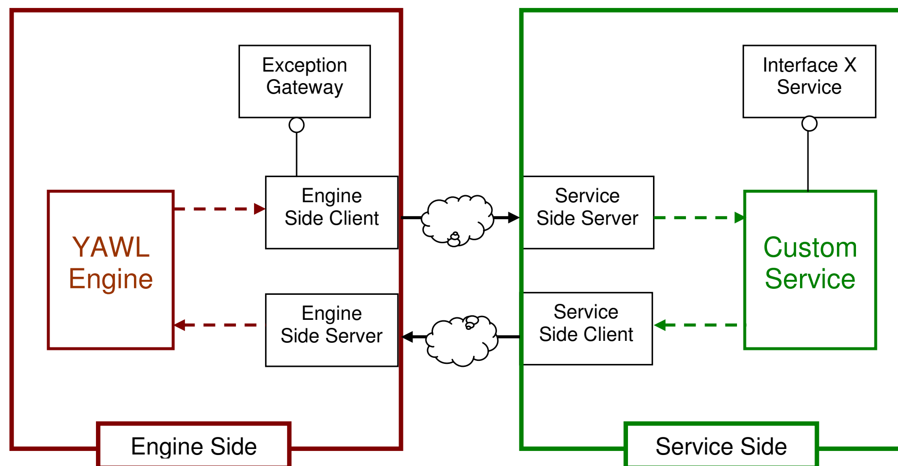

Figure 6.5: Interface X Architecture

The *ExceptionGateway* (java) interface describes the methods that the engine side client must implement to provide notifications to the custom service.

In fact, Interface X was created, as an output of this research, 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.

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 (or ignore) 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.

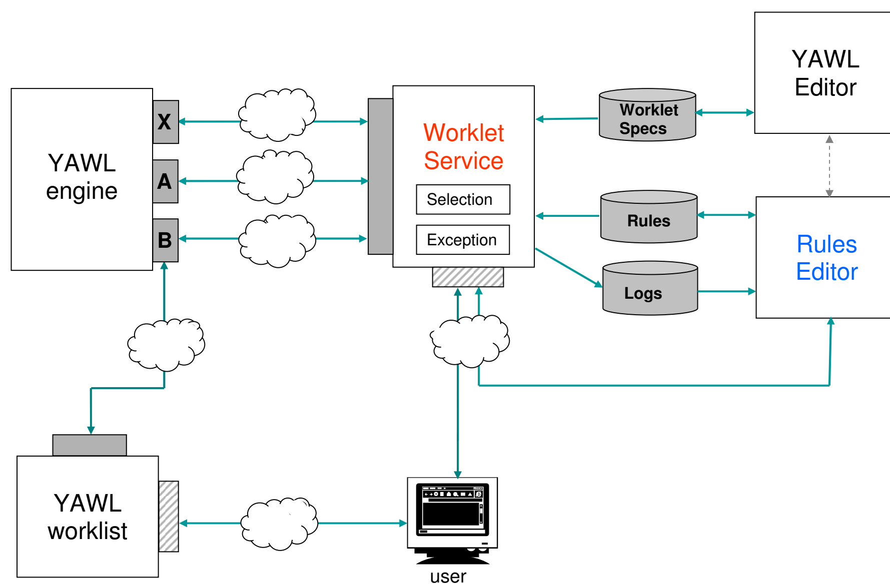

Figure 6.6: External Architecture of the Worklet Service

6.3 Worklet Service Architecture

This section describes the technical attributes and structure of the worklet service. The service is constructed as a web service and so consists of a number of J2EE classes and servlet pages, organised in a series of java packages.

The external architecture of the worklet service is shown in Figure 6.6. The entities 'Worklet Specs', 'Rules' and 'Logs' 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 (cf. Section 6.9), 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 6.9.2). The service also provides servlet pages that allow users to directly communicate with the service to raise external exceptions and to create and carry out administration tasks.

Any YAWL specification may have an associated rule set. The rule set for each specification is stored as XML data in a disk file that has the same name as the specification, but with an

```
        </ruleNode>
      </pre>
    </case>
  </constraints>
  <selection>
    <task name="Treat">
      <ruleNode>
        <id>0</id>
        <parent>-1</parent>
        <trueChild>1</trueChild>
        <falseChild>-1</falseChild>
        <condition>True</condition>
        <conclusion>null</conclusion>
        <cornerstone> </cornerstone>
        <description>root level default node</description>
      </ruleNode>
      <ruleNode>
        <id>1</id>
        <parent>0</parent>
        <trueChild>8</trueChild>
        <falseChild>2</falseChild>
        <condition>Fever = true</condition>
        <conclusion>
          <_1>
            <action>select</action>
            <target>TreatFever</target>
          </_1>
        </conclusion>
        <cornerstone>
```

Figure 6.7: Excerpt of a Rules File

".xrs" extension (signifying an 'XML Rule Set'). All rule set files are stored in the *rules* folder of the worklet repository. Figure 6.7 shows an excerpt from a rules file for a *Casualty Treatment* process, which shows detail of the first two rule nodes associated with the selection process for the 'Treat' task.

Figure 6.8 shows a representation of the internal architecture of the worklet service. The obvious hub of the service is the *WorkletService* class, which administrates the execution of the service and handles interactions with the YAWL engine via Interfaces A and B. The relationships between the various service objects in Figure 6.8 reflect a set of natural groupings into a series of java packages, as shown in Figure 6.9.

The discussion that follows in this section describes, in package order, each of the classes that makes up the worklet service. It should be read with reference to both Figures 6.8 and 6.9.

The outermost package is called *worklet*, and contains six sub-packages and one java class, *WorkletService*.

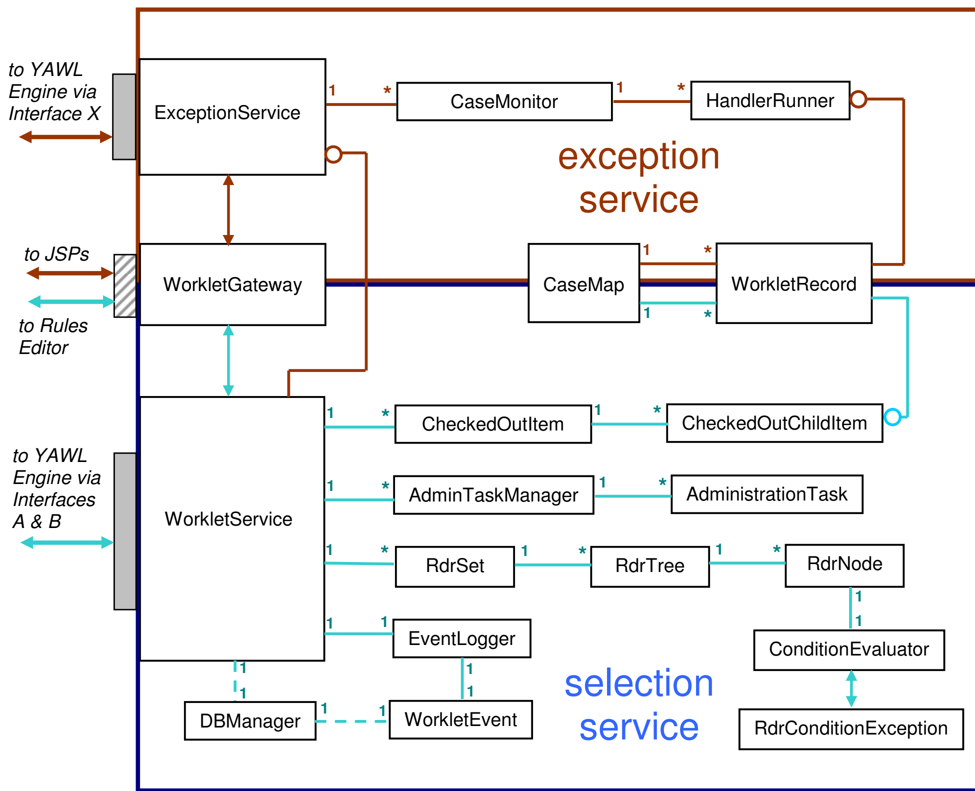

Figure 6.8: Internal Architecture of the Worklet Service

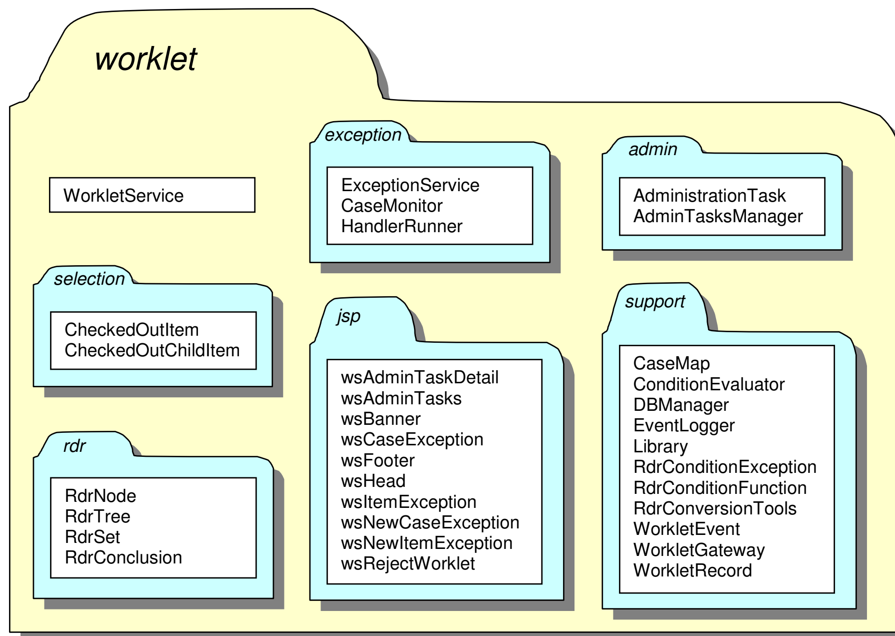

Figure 6.9: A Java Package View of the Worklet Service

6.3.1 The *WorkletService* class

The WorkletService class is the manager class for the selection and exception handling processes. For selection, it receives notification of an enabled workitem from the engine and attempts to substitute it with a worklet, as required. For exception handling, the ExceptionService class (see Section 6.3.3) extends from the WorkletService class and so leverages off the framework and functionality of that class. Figure 6.10 shows a UML Class Diagram for the WorkletService class.

From the fields listing it can be seen that this class manages the enumeration of exception types, connections through YAWL's Interfaces A & B to the engine, the loading and management of RDR rule sets, the checking in and out of enabled workitems and the loading, launching and replacing (if necessary) of worklets. It also oversees persistence of data for the service between sessions, and manages the administration tasks raised via user interaction with the servlet pages in the *jsp* package. Each of those features will be covered in more detail in the discussions of the relevant sub-packages.

6.3.2 The *selection* Package

The selection package contains two classes that store workitems and their related data between checking out of, and checking back in to, the YAWL engine. The WorkletService class may be managing a number of these objects at any one time.

- **CheckedOutItem**: The CheckedOutItem class represents a 'parent' workitem that has been checked-out of the engine; its purpose is to manage a dynamic set of data for each of the child items of which this workitem is a parent. The worklet service instantiates a CheckedOutItem object for a workitem that is checked out, and destroys it when the workitem (inclusive of its children) is checked back into the engine. In YAWL, all workitems consist of a least two objects: a parent workitem and one or more child workitems (both are instances of the YAWL *YWorkItem* class). An atomic task will be represented by a parent and one child; a multiple instance atomic task will have one parent and a number of children. When a workitem is checked out of the engine, the check-out process via Interface B actually checks out the parent and its first child, while additional child items (in the case of multiple atomic tasks) must be checked-out separately. Thus,

```
WorkletService
---------------------------------------------
+XTYPE_CASE_PRE_CONSTRAINTS: int = 0
+XTYPE_CASE_POST_CONSTRAINTS: int = 1
+XTYPE_ITEM_PRE_CONSTRAINTS: int = 2
+XTYPE_ITEM_POST_CONSTRAINTS: int = 3
+XTYPE_WORKITEM_ABORT: int = 4
+XTYPE_TIMEOUT: int = 5
+XTYPE_RESOURCE_UNAVAILABLE: int = 6
+XTYPE_CONSTRAINT_VIOLATION: int = 7
+XTYPE_CASE_EXTERNAL_TRIGGER: int = 8
+XTYPE_ITEM_EXTERNAL_TRIGGER: int = 9
+XTYPE_SELECTION: int = 10
#_user: String = "workletService"
#_password: String = "worklet"
#_adminUser: String = "admin"
#_adminPassword: String = "YAWL"
#_sessionHandle: String = null
#_engineURI: String = "http://localhost:8080/yawl"
-_workletURI: String = "http://localhost:8080/workletService/ib"
-_interfaceAClient: InterfaceA_EnvironmentBasedClient
-_handledParentItems: HashMap = new HashMap()
-_handledWorkItems: HashMap = new HashMap()
-_casesStarted: HashMap = new HashMap()
-_ruleSets: HashMap = new HashMap()
#_loadedSpecs: ArrayList = new ArrayList()
-_adminTasksMgr: AdminTasksManager = new AdminTasksManager()
#_persisting: boolean
#_dbMgr: DBManager
-_log: Logger
-_workletsDir: String
-_me: WorkletService
-_exService: ExceptionService
-restored: boolean = false
---------------------------------------------
<<create>>+WorkletService()
+getInstance(): WorkletService
#registerExceptionService(es: ExceptionService)
+completeInitialisation()
+handleEnabledWorkItemEvent(workItemRecord: WorkItemRecord)
+handleCancelledWorkItemEvent(workItemRecord: WorkItemRecord)
+handleCompleteCaseEvent(caseID: String, casedata: String)
+doGet(request: HttpServletRequest, response: HttpServletResponse)
-launchCase(specID: String, caseParams: String, sessionHandle: String, observer: boolean): String
-handleWorkletSelection(wir: WorkItemRecord)
-handleCompletingSelectionWorklet(caseId: String, wlCasedata: Element)
-ProcessWorkItemSubstitution(tree: RdrTree, coChild: CheckedOutChildItem)
-checkOutItem(wir: WorkItemRecord): CheckedOutItem
#checkOutParentItem(wir: WorkItemRecord): CheckedOutItem
#checkOutChildren(coi: CheckedOutItem)
#checkOutWorkItem(wir: WorkItemRecord): boolean
-checkInHandledWorkItem(coci: CheckedOutChildItem, wlCasedata: Element)
-checkinItem(wir: WorkItemRecord, in: Element, out: Element): boolean
#uploadWorklet(workletName: String): boolean
#launchWorkletList(wr: WorkletRecord, list: String): boolean
#cancelWorkletList(wr: WorkletRecord): boolean
#launchWorklet(wr: WorkletRecord, wName: String, setObserver: boolean): String
+replaceWorklet(itemid: String): String
-cancelWorkletCase(caseid: String, coci: WorkletRecord): boolean
#updateDataList(in: Element, out: Element): Element
-mapItemParamsToWorkletCaseParams(wr: WorkletRecord, wlName: String): String
-StringToElement(xmlString: String): Element
#getTree(specID: String, taskID: String, treeType: int): RdrTree
#RefreshRuleSet(specID: String)
+loadTree(specID: String)
+getDecompID(wir: WorkItemRecord): String
+getDecompID(specID: String, taskID: String): String
-getLoadedSpecs()
-getInputParams(specId: String): ArrayList
-iterateAllSpecsInputParams()
-attemptToGetChildDataList(w: WorkItemRecord)
-dump()
-iterateMap(map: HashMap)
-checkCacheForWorkItem(wir: WorkItemRecord)
+getXTypeString(xType: int): String
+getShortXTypeString(xType: int): String
-isUploaded(workletName: String): boolean
+isWorkletCase(caseID: String): boolean
+isAdminSession(sessionHandle: String): boolean
#connected(): boolean
-connectAsService(): String
-isRegisteredUser(user: String): boolean
+addAdministrationTask(caseID: String, title: String, scenario: String, process: String, taskType: int)
+addAdministrationTask(caseID: String, itemID: String, title: String, scenario: String, process: String, taskType: int)
+getAdminTaskTitles(): List
+completeAdminTask(adminTaskID: String)
+getAllAdminTasksAsList(): List
+getAdminTask(id: String): AdministrationTask
-restoreDataSets()
-restoreHandledParentItems(): HashMap
-restoreHandledChildItems(): HashMap
-restoreAdminTasksManager(): AdminTasksManager
```

Figure 6.10: The WorkletService Class

the CheckedOutItem class keeps track of its checked-out 'children'.

- **CheckedOutChildItem**: The CheckedOutChildItem class maintains a dataset for a 'child' workitem that has been checked-out of the engine (via its parent). Each checked-out child item has its own discrete datalist, which may be used to launch a different substitute worklet for each child (when the workitem is an instance of a multiple instance atomic task). A number of CheckedOutChildItem instances (representing the child workitems) are maintained by one CheckOutItem (representing the parent workitem). A CheckedOutChildItem object also records the name and case identifier of the worklet that has been substituted for it.

6.3.3 The *exception* Package

The exception package contains three classes that manage the exception sub-service.

Management of the whole exception handling process is carried out by an instance of the ExceptionService class (like the WorkletService class, only one instance of this class is created at runtime). This class extends from WorkletService, and so inherits from it the methods for the management of rule sets, the launching of worklets, connections to the engine and so on. The ExceptionService class manages the handling of exceptions that may occur during the life of a case instance. It receives notification of events from the engine via YAWL's *InterfaceX* at various milestones for constraint checking, and when exceptional events occur. It oversees the launching and execution of exception handlers (exlets) when required, which may involve the running of compensatory worklets.

Figure 6.11 shows that this class has a relatively large number of methods that are required to manage the exception handling process, including methods for handling event notifications from the engine, checking constraints at the case and workitem level, raising exceptions as required, searching the RDR rule sets for appropriate handlers, executing the exlets returned from the rule set, and taking the necessary action as defined in the primitives of the executed exlet.

The two other classes in this package directly support the operations of the ExceptionService class:

- **CaseMonitor**: On notification from the YAWL engine that a new case instance has

ExceptionService

-_handlersStarted: Map = new HashMap()
-_monitoredCases: Map = new HashMap()
-_log: Logger
-_ixClient: InterfaceX_ServiceSideClient
-_me: ExceptionService
-_pushedItemData: WorkItemConstraintData
-_exceptionURI: String = "http://localhost:8080/workletService/ix"
-mutex: Object = new Object()

<<create>>+ExceptionService()
+getInst(): ExceptionService
+completeInitialisation()
+handleCaseCancellationEvent(caseID: String)
+handleCheckWorkItemConstraintEvent(wir: WorkItemRecord, data: String, preCheck: boolean)
+handleCheckCaseConstraintEvent(specID: String, caseID: String, data: String, preCheck: boolean)
+handleTimeoutEvent(wir: WorkItemRecord, taskList: String)
+handleWorkItemAbortException(wir: WorkItemRecord)
+handleResourceUnavailableException(wir: WorkItemRecord)
+handleConstraintViolationException(wir: WorkItemRecord)
-checkConstraints(monitor: CaseMonitor, pre: boolean)
-checkConstraints(monitor: CaseMonitor, wir: WorkItemRecord, pre: boolean)
-getExceptionHandler(monitor: CaseMonitor, taskID: String, xType: int): RdrConclusion
-raiseException(cmon: CaseMonitor, conc: RdrConclusion, sType: String, xType: int)
-raiseException(cmon: CaseMonitor, conc: RdrConclusion, wir: WorkItemRecord, xType: int)
-processException(hr: HandlerRunner)
-doAction(runner: HandlerRunner): boolean
#launchWorkletList(hr: HandlerRunner, list: String): boolean
-doContinue(runner: HandlerRunner)
-doSuspend(runner: HandlerRunner)
-doRemove(runner: HandlerRunner)
-handleCompletingExceptionWorklet(caseId: String, wlCasedata: Element)
-suspendWorkItem(hr: HandlerRunner): boolean
+suspendWorkItem(itemID: String): boolean
-suspendWorkItemList(items: List): boolean
+suspendCase(caseID: String): boolean
-suspendCase(hr: HandlerRunner): boolean
-getListOfExecutingWorkItems(scope: String, id: String): List
-getListOfSuspendableWorkItems(scope: String, id: String): List
-getListOfSuspendableWorkItemsInChain(caseID: String): List
-suspendAllCases(hr: HandlerRunner): boolean
-suspendAncestorCases(runner: HandlerRunner): boolean
-removeWorkItem(wir: WorkItemRecord)
-removeCase(hr: HandlerRunner)
-removeCase(caseID: String)
-removeAllCases(specID: String)
-removeAncestorCases(runner: HandlerRunner)
-getFirstAncestorCase(runner: HandlerRunner): String
-forceCompleteWorkItem(wir: WorkItemRecord, data: Element)
-restartWorkItem(wir: WorkItemRecord)
-failWorkItem(wir: WorkItemRecord)
-unsuspendWorkItem(wir: WorkItemRecord): WorkItemRecord
-unsuspendList(runner: HandlerRunner)
-updateWIR(wir: WorkItemRecord): WorkItemRecord
-updateItemData(runner: HandlerRunner, wlData: Element)
-updateCaseData(runner: HandlerRunner, wlData: Element)
-cancelLiveWorkletsForCase(monitor: CaseMonitor)
-executeWorkItem(wir: WorkItemRecord): CheckedOutItem
-isExecutingItemException(xType: int): boolean
+isCaseLevelException(xType: int): boolean
-getLiveWorkItemsForIdentifier(idType: String, id: String): List
-getWorkItemRecordsForTaskInstance(specID: String, taskID: String): List
-getIntegralID(id: String): String
-registerThisAsExceptionObserver()
-destroyMonitorIfDone(monitor: CaseMonitor, caseID: String)
-completeCaseMonitoring(monitor: CaseMonitor, caseID: String)
-startItem(wir: WorkItemRecord): List
+getWorkItemRecord(itemID: String): WorkItemRecord
+getSpecIDForCaseID(caseID: String): String
+getExternalTriggersForCase(caseID: String): List
+getExternalTriggersForItem(itemID: String): List
-getExternalTriggers(tree: RdrTree): List
-getConditionValue(cond: String, var: String): String
+raiseExternalException(level: String, id: String, trigger: String)
+replaceWorklet(xType: int, caseid: String, itemid: String, trigger: String): String
+isWorkletCase(caseID: String): boolean
+getStatus(taskName: String): String
-restoreDataSets()
-restoreRunners(): HashMap
-restoreMonitoredCases(runnerMap: HashMap): HashMap
-rebuildHandlersStarted(runners: List)
-pushCheckWorkItemConstraintEvent(wir: WorkItemRecord, data: String, preCheck: boolean)
-popCheckWorkItemConstraintEvent(mon: CaseMonitor)

Figure 6.11: The ExceptionService Class

been launched (via a PreCaseConstraint notification), the ExceptionService creates a new CaseMonitor object that stores information about the case relevant to the service, and it remains active for the life of the case instance. The primary purpose is to minimise calls across the interface back to the engine while a case is active. Its other main role is to manage the descriptors of raised exlets while they are executing by maintaining a set of HandlerRunner objects.

- **HandlerRunner**: The HandlerRunner class manages a single exlet. An instance of this class is created for each exception process raised by a case. The CaseMonitor class (above) maintains the current set of HandlerRunners for a case (at any one time, a case may have a number of active HandlerRunner instances). This class also manages an executing worklet instance running as a compensation for a 'parent' case when required. The key data member of this class is an instance of RdrConclusion (cf. Section 6.3.4), which contains the sequential set of exception handling primitives for the particular exlet managed by this object. The role that a HandlerRunner object performs for the exception service class shares many similarities with the role a CheckOutChildItem object carries out within the selection service. As a result of those related responsibilities, both the HandlerRunner and CheckedOutChildItem classes are derived from the base WorkletRecord class (cf. Section 6.3.6).

6.3.4 The *rdr* Package

The classes in the rdr package support the loading, storing, constructing, searching and evaluation of the ripple down rules used by the service.

- **RdrNode**: This class is an implementation of a ripple down rule node. Each RdrNode contains an individual conditional expression, a conclusion and associated data parameters. Each RDRTree object (described below) maintains a set of these nodes that comprise a rule tree. The class has data members that store references to the parent node and child nodes (if any), a node identifier and members for the conditional expression, the conclusion to be returned (conceptually an exlet structure but stored externally as XML and internally as a JDOM Element data type) and the cornerstone case data for the node (also stored as XML externally and as a JDOM Element internally).

This class contains the recursive search method which locates and returns the appropriate node based on the current case data and the traversal of the conditional expressions of each node. In fact, a pair of RdrNodes is returned by this method: (i) the last satisfied node, representing the result of this search and containing the worklet or exlet to execute, and (ii) the last node searched, needed when the resultant worklet or exlet of the search is rejected (by a user) as inappropriate for a particular case and a new rule formulated — the newly created rule node containing the new rule will be added as a child to the last searched node, via the algorithm described in Section 4.2. Below is the search method in its entirety:

```
/**
*  Recursively search each node using the condition of each to determine
*  the path to take through the tree of which this node is a member.
*  @param caseData - a JDOM Element that contains the set of data
*         attributes and values that are used to evaluate the conditional
*         expressions
*  @param lastTrueNode - the RdrNode that contains the last satisfied
*         condition
*  @return a two node array: [0] the last satisfied node
*                            [1] the last node searched
*/

public RdrNode[] recursiveSearch(Element caseData, RdrNode lastTrueNode){
    RdrNode[] pair = new RdrNode[2];

    ConditionEvaluator ce = new ConditionEvaluator() ;

    try {
        if(ce.evaluate(condition, caseData)){ // if condition evals to True
            if(trueChild == null) {           // ...and no exception rule
                pair[0] = this;               // this is last satisfied
                pair[1] = this;               // and last searched
            }
            else {                            // test the exception rule
                pair = trueChild.recursiveSearch(caseData, this);
            }
        }
        else {                                // if condition evals to False
            if(falseChild == null){           // ...and no if-not rule
                pair[0] = lastTrueNode;       // pass on last satisfied
                pair[1] = this;               // and this is last searched
            }
            else{                             // test the next if-not rule
                pair = falseChild.recursiveSearch(caseData,lastTrueNode);
            }
        }
    }
    catch( RdrConditionException rde ) {      // bad condition found
            _log.error("Search Exception", rde) ;
            pair[0] = null ;
            pair[1] = null ;
  }
```

```
  return pair ;
}
```

- **RdrTree**: This class maintains a set of RdrNodes, which together comprise one rule tree, internally structured as a linked list. Each RdrTree contains the set of rules for one particular exception (or selection) type and, in the case of item-level exception types, for one particular task in a specification. Thus, a specification may have a number of rule trees defined for it. In addition, the class records which specification and task (if any) this tree is constructed for, and begins the search process (from the root node).

- **RdrSet**: The RdrSet class manages the set of RdrTrees for an entire specification. For each case-level exception type, one tree is managed. For each item level exception type, a number of trees is managed — one for each task. However, it is not necessary (in fact, very unlikely) to have defined a tree for each task for each exception type; only those exception and/or task combinations that are required to have exceptions handled and/or selections made need be defined. This class is also responsible for loading rule sets from file into the set-tree-node hierarchy, translating from XML to RdrNode via JDOM constructs. Once a rule set has been loaded, it remains so for the life of the worklet service's instantiation, but may be updated if new rules are added during that time. Thus at any time, the service will call this class to return a particular tree. If the requested tree has not yet been loaded, it will automatically load the entire rule set for the specification requested.

- **RdrConclusion**: This class stores a conclusion returned from a selected RdrNode as the result of a rule tree search. The conclusion is a JDOM Element, representing an exlet, and consisting of a number of children of the form:

  ```
  <_n>
    <action>someAction</action>
    <target>someTarget</target>
  <_n>
  ```

  where *n* is an ordinal number that indicates an ordering or sequence of the primitives that form the conclusion. From this it can be seen that an exlet is stored in XML format and that each primitive of the exlet can be described using and 'action-target' pair (i.e. perform some action against some target). A full description of the types of actions and targets that can be stored in an RdrConclusion object can be found in Section 6.6.

A search of a rule tree returns a "null" conclusion when none of its node's conditions evaluate to true (besides the root node) — that is, whenever no rule is satisfied for the particular exception (or selection) type based on the context of the case instance. If a "null" result occurs as part of a selection traversal, then no worklet substitution occurs and the task is allowed to execute as an 'ordinary' YAWL task; if it occurs as part of an exception-handling traversal, the exception notification is simply ignored (exception notifications are also ignored if there is no rule tree defined for that exception type in the rule set). Thus, an RdrConclusion object is created as the result of a rule search only if something other than "null" is returned by the search — i.e. a worklet or exlet has been contextually identified. The ExceptionService instance tests the value of the RdrConclusion's 'conclusion' data member to determine if an exception has occurred and, if so, begins execution of the exlet stored in this RdrConclusion object.

If a valid RdrConclusion is returned from a rule search, a HandlerRunner object is instantiated that stores the RdrConclusion. From the HandlerRunner, the ExceptionService instance retrieves each primitive of the exlet in turn and takes the specified action and performs it against the specified target (of the particular case). As each primitive's action completes, the ExceptionService retrieves and performs the next primitive, until such time that the exlet has completed. The sequential delivery of primitives is maintained by the HandlerRunner object.

6.3.5 The *admin* Package

This package manages the creation, delivery and storage of administration tasks raised by users who have either rejected a worklet as an inappropriate selection for the current case instance, or who have raise an unexpected external exception (that is, an external exception the cause of which has not been previously defined). It consists of two classes:

- **AdministrationTask**: This class stores the details of a pending administration task, which may be viewed via the worklet admin screens (cf. Section 6.3.7). An administration task is raised by the user when one of three events occur:
  - the rejection of a worklet selection (either resulting from a selection service substitution or an exception service compensation);

  - the raising of an unexpected (i.e. previously undefined) case-level external exception; or
  - the raising of an unexpected (i.e. previously undefined) workitem-level external exception.

  An AdministrationTask object acts as a data set for a user-supplied series of free-text values, including a description of the event, a description of a suggested process required to handle the event (thus capturing 'sub-domain' or local knowledge of how the work is to be performed), and a suggested name for the new worklet. An administrator will use the supplied descriptions to create a new rule and worklet to add to the rule set for that specification based on the new context described by the case instance.

- **AdminTasksManager**: The AdminTasksManager class manages the current set of administration tasks for an installation of the service — that is, outstanding tasks that an administrator is to attend to. The task set is managed through a dedicated servlet page, which allows the administrator to view a task and remove it from the set when it has been completed.

The WorkletService class actually maintains the current AdminTasksManager object and provides a number of 'interface' methods which act as a conduit between the relevant jsps and the current set of administration tasks.

6.3.6 The *support* Package

The support package has a number of classes that are used to support the functionality and operation of the worklet service. Each class serves a specific purpose:

- **ConditionEvaluator**: This class is at the heart of the evaluation of the conditional expressions contained in the rule nodes. It is used by the RdrNode class to evaluate its own condition and thus allows the rule traversal to occur. It receives one or two arguments for each evaluation: a conditional expression from the rule node (provided as a String value); and an optional datalist argument, provided as a JDOM Element value — if supplied, it will be used to retrieve values for any variable names used in the expression. The ConditionEvaluator object will parse the String-based expression and tokenise it into its

appropriate operands (converted to correct data types) and operators, and evaluate it to a boolean value.

The expression may contain any of the following operators:

- Arithmetic: $*$ $/$ $+$ $-$
- Comparison: $=$ $!=$ $>$ $<$ $>=$ $<=$
- Logical: $\&$ $||$ $!$

which observe the usual order of precedence including support for the use of parentheses to group sub-expressions. Operands may be numeric literals, string literals, variable names or function names.

A specific RdrConditionException will be raised if the expression is malformed or does not evaluate to a boolean value. The recursive search function of the RdrNode class creates a ConditionEvaluator instance each time a tree is traversed.

- **DBManager**: This class provides persistence support for the entire worklet service. The service is persisted using the *Hibernate* package, an open-source object/relational persistence and query API. Service objects are serialised and stored in a relational database, against which queries can be run using a portable SQL extension language. The DBManager class abstracts this functionality via a small number of interface-like methods that are used throughout the worklet service. The class will also create the required database tables if they are not already extant. Worklet service persistence may be enabled or disabled via a configuration file setting (see Section 6.4).

- **EventLogger**: The EventLogger class manages the writing of service events to an event log. The events that are logged include the checking in and out of workitems, and the launch, completion and cancellation of worklets (events regarding the execution of the worklet as a YAWL instance are captured by the YAWL engine logger, as are actions that change the state of an executing instance — that is, logging roles are not duplicated between engine and service). Events are written to a database table via the DBManager object if persistence is configured as enabled, or to a comma-separated file in the worklet repository if it is not, ensuring that a log is always generated and stored. Note that the operation of the service itself, as opposed to the operations of worklets, is also logged

to an audit file in the host web server's logs (an example of such a file can be found in Appendix C).

- **Library**: This class contains a set of generic static methods for use throughout the service. The methods provided include support for setting and getting the disk path to the worklet repository, the enabling/disabling of persistence as read from the configuration file, and various String conversion methods.
- **RdrConditionException**: An RdrConditionException is thrown by a ConditionEvaluator object when an attempt is made to evaluate a rule's condition and it is found to be malformed or does not evaluate to a boolean result. It is extended from the general java Exception class.
- **RdrConditionFunctions**: This class provides developers with a 'hook' into a ConditionEvaluator object via the definition of functions that can then be referred to in the conditional expressions of rule nodes. It therefore extends the capabilities of a rule's condition so that data can be sourced and queried external to the service and the current case (for example, through the querying of process logs and resource allocations). This class can be designed to be easily extended and recompiled without the need to recompile the entire service. See Section 6.8 for more information on the use of this class in practice.
- **RdrConversionTools**: This class contains a number of static methods that convert some service objects to Strings and vice versa. It also supports the stringifying of some objects for persistence purposes, particularly the translation of RdrNode and (YAWL's) WorkItemRecord objects to Strings and vice versa.
- **WorkletEvent**: The sole purpose of this class is to generate an event log record for use via persistence. For each event to be logged, an instance is created via the constructor of the EventLogger class, then the object is persisted to create one event log record.
- **WorkletGateway**: The WorkletGateway class acts as a gateway between the worklet service and the external Rules Editor. It provides functionality to trigger a running worklet replacement due to an addition to the ruleset (by the editor) and also initialises the worklet service on startup with values from the service's configuration file.
- **WorkletRecord**: This class maintains a generic dataset for derived classes that manage a currently running worklet or exlet for a 'parent' process. It is the base class of the

CheckedOutChildItem and HandlerRunner classes. It also produces a 'log' file after each rule tree traversal containing attributes describing the result of the search and the current case and workitem identifiers, to be used by the Rules Editor when adding a new rule for this case following a worklet rejection.

6.3.7 The *jsp* Package

The worklet service uses a set of java servlet pages (jsps) to provide users with a browser-enabled interface which allows access to some of the service's functionality.

When the exception service is enabled (via the configuration file) three buttons are added to a user's inbox in YAWL's worklist handler that provide the functionality to:

1. reject a worklet (as an inappropriate choice for the current case instance based on its context);
2. raise an *unexpected* case-level external exception (that is, for an event other than those listed in the specifcation's rule set); and
3. raise an *unexpected* workitem-level external exception.

A user's direct interaction with the worklet service, instigated via the buttons provided to the user interface, is achieved using the appropriate series of the worklet service jsps below.

- **wsBanner, wsHead and wsFooter**: These sub-pages provide the header and footer information for each of the other worklet jsps. The header is similar to the one presented in the YAWL pages (for user familiarity), but includes a worklet logo to allow the user to differentiate between YAWL pages and worklet service pages. An example of the header with the worklet service logo (or identity displayed can be seen in the left-side of the banner in Figure 6.12.
- **wsAdminTasks**: The Admin Tasks page retrieves the set of currently outstanding administration tasks via the worklet service and displays them as a list. From the list (which shows the admin task identifier, the identifier of the originating case, the task title and type) an administrator can select and view the details of a particular task (via the wsAdminTaskDetail page) and can mark an admin task as completed.

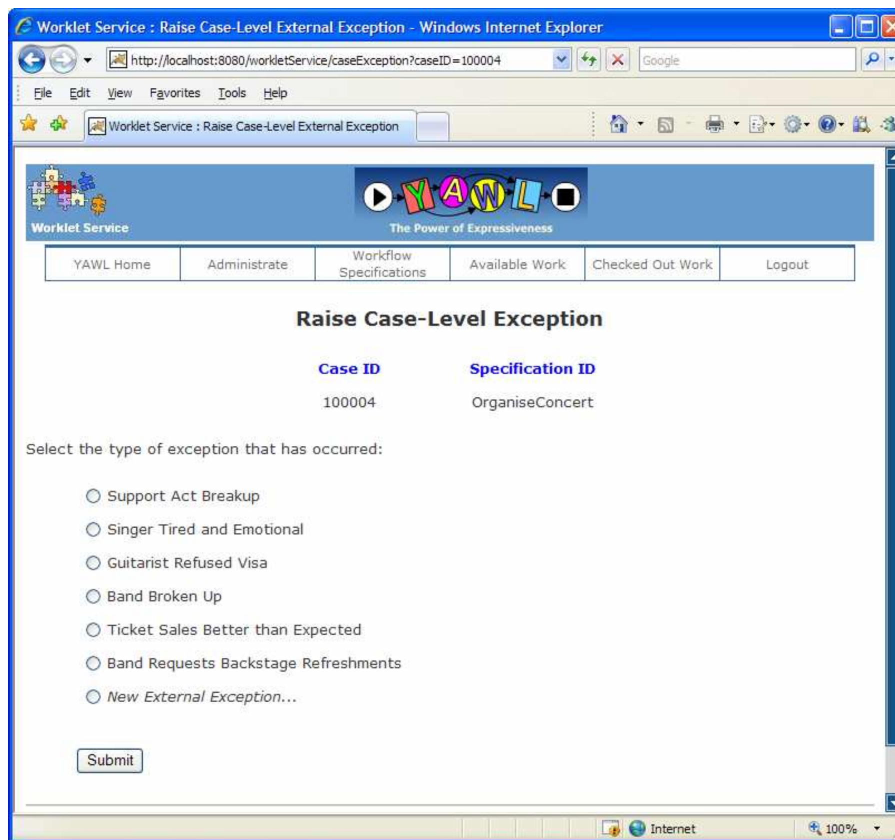

Figure 6.12: The Raise Case Level External Exception Page

- **wsAdminTaskDetail**: This page displays to an administrator the details of a particular admin task (selected from the wsAdminTasks page), including the title, scenario and a suggested process, in free text as defined by the user. From the details on this page, an administrator will invoke the rules editor to create the requested new worklet, and/or exlet for an unexpected exception, as required.

- **wsCaseException**: This page, as seen in Figure 6.12, allows the user to raise a new case level exception. The user is shown a list of 'triggers' (short text describing the exception) for all of the case level external exceptions currently defined in the rule set for the particular specification, from which the user may select the appropriate trigger for the current event, if available. Making a selection from the list of available triggers will launch the corresponding exlet, after taking the current case context into account via the case-level external exception rule tree. If the exception trigger is not listed on this page, it is by definition an unexpected exception, which the user may indicate by choosing the 'New External Exception' option at the bottom of the list. This selection will display the wsNewCaseException page to allow the user to describe the exception, suggest a possible

handling process (as free text), and submit the task to the administrator.

- **wsNewCaseException**: A user enters into this page the necessary details (as free text) to define a new case-level external exception. This page is invoked from the wsCaseException page when none of the listed (i.e. expected) case-level external exception triggers meet the circumstances of the current event. When a user completes the decription on this page, it becomes a new administration task and is added to the list of current tasks for the administrator to perform.

- **wsItemException, wsNewItemException**: These two pages have similar functionality to their case-level equivalents above, but are used to select and or raise an item-level external exception. The wsItemException page is invoked via a button on the 'Checked Out Work Items' page of YAWL's worklist handler, rather than the 'Case Viewer' worklist page used by the case-level exception pages.

- **wsRejectWorklet**: As the name implies, this page provides the user with the ability to reject a chosen worklet that is deemed inappropriate for the particular case instance, based on its context. The page is similar to the 'New Exception' pages in that it collects from the user the details of the rejection, and then submits it as a new admin task to the administrator for action.

6.4 Service Installation and Configuration

Like other YAWL Custom Services, the worklet service is a java-based web service and so must be hosted on a web server. By default, an installation of the open-source web server *Apache Tomcat* is used to host the YAWL environment, and as a consequence the worklet service is also hosted on Tomcat. Further, since the worklet service is distributed as part of the YAWL installation, it resides within the same host (again by default).

However, since it is a totally discrete service, the worklet service may be installed on and hosted by any remote web server. In fact, it is possible for one instance of the worklet service to manage the selection and exception handling requirements of a number of YAWL engines across domains simultaneously (or potentially other types of enactment engines that conform to the interface).

The worklet service is distributed as a java web archive (or *.war* file), which includes compiled versions of all of the classes of the service, ancillary classes, the worklet repository (with samples) and the Rules Editor. The service is installed by placing the war file in the appropriate applications folder of the web server.

Each web service, including the YAWL engine, makes use of a configuration file (in Tomcat terminology, a "deployment descriptor"), which sets the parameters for the instantiation and operation of the service instance. Individual YAWL services use these parameters to specify settings such as the location of services, enabling of the exception service, enabling of persistence, location of the worklet repository, and so on. Setting values for parameters can be done using any text editor.

This section discusses the various parameters that can or may be set to successfully deploy the worklet service.

6.4.1 Configuration — Service Side

The configuration file for the worklet service has parameters for specifying the URL of the YAWL Engine's Interface B, the location of the WorkletService and ExceptionService classes (i.e. the relative location of those classes within the web archive), the locations of the interface classes that send notifications from the engine to the service, and mappings for resolving the URLs of the service's jsps to the internal location of the page being requested. All of these parameters are set to their defaults in the deployed web archive, which assumes the worklet service is installed on the same web server as the YAWL engine, but of course these can be changed to their relevant values where the service is installed remotely to the engine.

The configuration file also has two parameters that must be set for each individual installation (see Figure 6.13):

- the *Repository* parameter, which maps the disk location of the worklet repository for that installation, allowing the repository to be installed in an appropriate place of the administrator's choosing depending on local needs; and
- the *EnablePersistence* parameter, which when set to *true* enables persistence of instantiated service objects to a database so that items and cases currently being handled by the service may be persisted across sessions, and to allow log records to be written to a

```
<context-param>
   <param-name>Repository</param-name>
   <param-value>C:\Worklets\repository\</param-value>
   <description>
      The path where the worklet repository is installed.
   </description>
</context-param>

<context-param>
   <param-name>EnablePersistence</param-name>
   <param-value>false</param-value>
   <description>
      'true' to enable persistence and logging
      'false' to disable
   </description>
</context-param>
```

Figure 6.13: The Worklet Service Configuration File (detail)

```
<!-- PARAMS FOR EXCEPTION SERVICE -->

<context-param>
   <param-name>EnableExceptionService</param-name>
   <param-value>true</param-value>
   <description>
      Set this value to 'true' to enable monitoring by an
      Exception Service (specified by the URI param below).
      Set it to 'false' to disable the Exception Service.
   </description>
</context-param>
```

Figure 6.14: The YAWL Engine Configuration File (detail)

database table. If the parameter is set to false, service states are not persisted, and log entries are instead written to a comma-delimited file in the worklet repository.

6.4.2 Configuration — Engine Side

When the worklet service has been installed, it is automatically configured to receive notifications from the engine for worklet-enabled tasks as part of the selection process — that is, there are no explicit configuration tasks necessary to enable the selection service. For the exception service, there are two relevant parameters in the engine's configuration file; the first sets the URL of the exception service (set by default to a location relative to the same web server as the engine). The second parameter enables or disables the service (Figure 6.14); when the parameter is set to *true*, the engine notifies the service at various points when exceptions have (or may have) occurred throughout the life cycle of every case launched in the engine.

In addition to the parameter settings in the engine's configuration file, the exception service

```
<!-- This param, when available, enables the worklet exception
     service add-ins to the worklist. If the exception service
     is enabled in the engine, then this param should also be
     made available. If it is disabled in the engine, the
     entire param should be commented out. -->
<!--
<context-param>
    <param-name>InterfaceX_BackEnd</param-name>
    <param-value>http://localhost:8080/workletService</param-value>
    <description>
        The URL location of the worklet exception service.
    </description>
</context-param>
-->
```

Figure 6.15: The YAWL Worklist Configuration File (detail)

makes use of extensions (or 'hooks') built into the YAWL worklist handler to provide methods for interacting with the service (for example allowing the raising of an external exception); so, if the exception service is enabled via the engine's configuration file, then the extensions to the worklist must also be enabled via its configuration file. The worklist handler has been deployed as a discrete service also, so has its own configuration file. Within that file, a parameter is supplied to specify the URL of the exception service; by default it is set to a URL of the service relative to the same web server as the engine, but again it can be modified if the exception service is installed remotely. Also by default, the parameter is commented out, since the exception service is disabled by default in the engine's configuration when first deployed. When the comment tags are removed, the worklist becomes aware that the exception service is enabled and so enables the appropriate methods and items available via its user interface. See Figure 6.15 for the relevant part of the worklist's configuration file.

6.5 Worklet Process Definition

Fundamentally, a worklet is nothing more than a YAWL process specification that has been designed to perform one part of a larger, parent specification. However, it differs from a decomposition or sub-net in that it is dynamically assigned to perform a particular task at runtime, while sub-nets are statically assigned at design time. So, rather than being forced to define all possible branches in a specification, the worklet service allows the definition of a much simpler specification that will evolve dynamically as more worklets are added to the repertoire for particular tasks within it.

Figure 6.16 shows a simple example specification (in the YAWL Editor) for a Casualty Treatment process. Note that this process specification has been intentionally simplified to

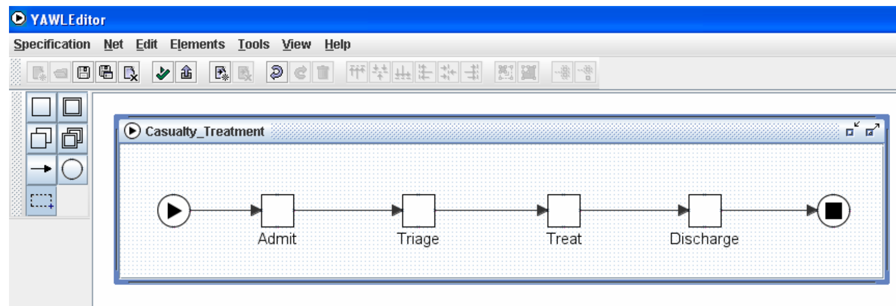

Figure 6.16: Parent 'Casualty Treatment' Process

demonstrate the operation of the worklet service; while it is not intended to portray a realistic process, it is desirable to not camouflage the objective of this demonstration by using a more complex process specification.[2]

The process model in Figure 6.16 is an ordinary YAWL process, which, without interaction from the worklet service, would remain completely static. However, the *Treat* task is worklet-enabled, and so will be substituted at runtime with the appropriate worklet based on the patient data collected in the *Admit* and *Triage* tasks. That is, depending on each patient's actual physical data and reported symptoms, the service will select and launch, in place of the task, the worklet that best treats the patient's condition.

In this example, only the *Treat* task is worklet-enabled; the other tasks will be handled directly by the YAWL environment. So, when a Casualty Treatment process is executed, the YAWL Engine will notify the worklet service when the *Treat* task becomes enabled. The worklet service will then examine the data of the task and use it to determine which worklet to execute as a substitute for the task.

Since a worklet specification is a standard YAWL process specification, it is created in the YAWL Editor in the usual manner. Figure 6.17 shows a very simple example worklet to be substituted for the *Treat* top-level task when a patient presents with a fever.

In itself, there is nothing special about the Treat Fever specification in Figure 6.17. Even though it will be considered by the worklet service as a member of the worklet repertoire for the *Treat* task and may thus be considered a "worklet", it also remains a standard YAWL specification and as such may be executed directly by the YAWL engine without any reference to the worklet service, if desired.

The association of tasks with the worklet service is not restricted to top-level specifications.

[2]In contrast, Chapter 7 examines two real-world case studies that are used to validate the worklet approach.

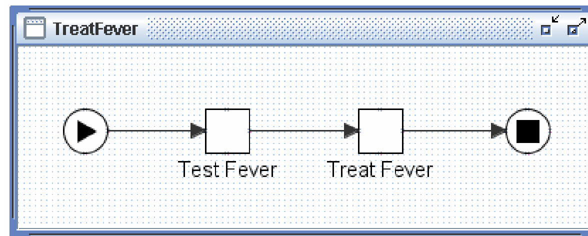

Figure 6.17: The 'Treat Fever' Worklet Process

Worklet specifications also may contain tasks that are associated with the worklet service and so may have worklets substituted for them, so that a hierarchy of executing worklets may sometimes exist. It is also possible to recursively define worklet substitutions — that is, a worklet may contain a task that, while certain conditions hold true, is substituted by another instance of the same worklet specification that contains the task.

Any number of worklets can be created for a particular task. For the Casualty Treatment example, there are five example worklets in the repertoire for the *Treat* task, one for each of the five conditions that a patient may present with in the *Triage* task: Fever, Rash, Fracture, Wound and Abdominal Pain. Which worklet is chosen for the *Treat* task depends on which of the five is given a value of True in the *Triage* task.

The *Treat* task is associated with the worklet service (that is, the task is 'worklet-enabled') via the YAWL Process Editor's, *Update Task Decomposition* dialog (Figure 6.18). This dialog shows the variables defined for the task — each one of these maps to a net-level variable, so that in this example all of the data collected from a patient in the first two tasks are made available to this task. The result is that all of the relevant current case data for this process instance can be used by the worklet service to enable a contextual decision to be made. Note that it is not necessary to map all available case data to a worklet enabled task, only that data required by the service to make an appropriate decision via the conditional expressions in the rule nodes defined for the task. The list of task variables in Figure 6.18 shows that most are defined as 'Input Only', indicating that the values for those variables will not be modified by any of the worklets that may be executed for this task; they will be used only for the selection process. The last three variables are defined as 'Input & Output', so that the worklet can modify and return (i.e. map back) to these variables data values that are captured during the worklet's execution. The dialog includes a section at the bottom called *YAWL Registered Service Detail*. It is here that the task is associated with the worklet service (i.e. made worklet-enabled) by choosing the

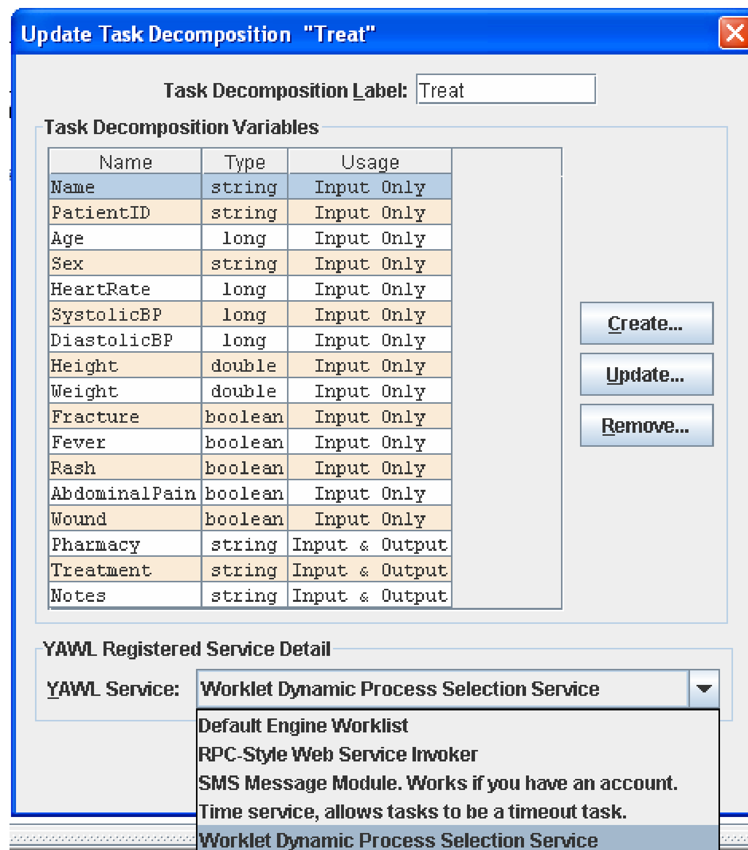

Figure 6.18: The Update Task Decomposition dialog for the Treat task

worklet service from the list of available services.

Those data values that are required to be mapped from the parent task to the worklet need to be defined as net-level variables in each of the worklet specifications made available to this task. Figure 6.19 shows the net-level variables for the TreatFever worklet.

Note the following:

- Only a subset of the variables defined in the parent *Treat* task (see Figure 6.18) are defined here. It is only necessary to map from the parent task to the worklet those variables that contain values to be displayed to the user, and/or those variables that the user will supply values for to be passed back to the parent task when the worklet completes;
- The definition of variables is not restricted to those defined in the parent task. Any additional variables required for the operation of the worklet may also be defined here; their values will not be passed back to the parent task when the worklet completes;
- Only those variables that have been defined with an identical name and data type to variables in the parent task and with a *Usage* of 'Input Only' or 'Input & Output' will have

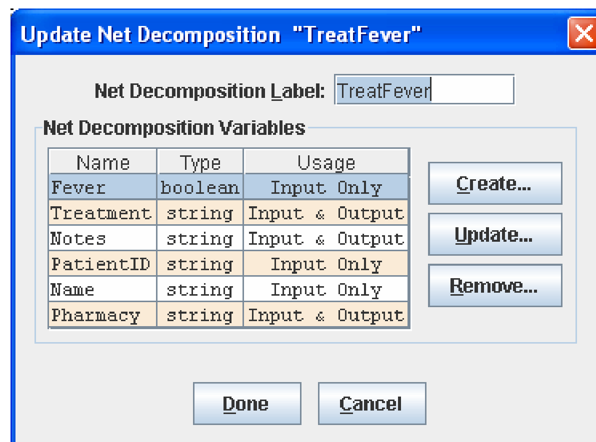

Figure 6.19: The Net-level variables for the TreatFever worklet

data passed into them from the corresponding variables of the parent task by the worklet service when the worklet is launched; and

- Only those variables that have been defined with an identical name and data type to variables in the parent task and with a *Usage* of 'Output Only' or 'Input & Output' will have their data values mapped back to the corresponding variables of the parent task by the worklet service when the worklet completes.

For this example (Figure 6.19), it can be seen that the values for the PatientID, Name and Fever variables will be used by the TreatFever worklet as display-only values; the Notes, Pharmacy and Treatment variables will receive values during the execution of the worklet and will map those values back to the top-level Treat task when the worklet completes.

6.6 Exlet Process Definition

This section discusses the definition of exlets. It first introduces the seven different types of process exception that are supported by the current version of the worklet exception service. It then describes each of the handling primitives that may be used to form an exlet.

The three exception types not yet supported are:

- ItemAbort - occurs when a workitem being handled by an external program reports that the program has aborted before completion;

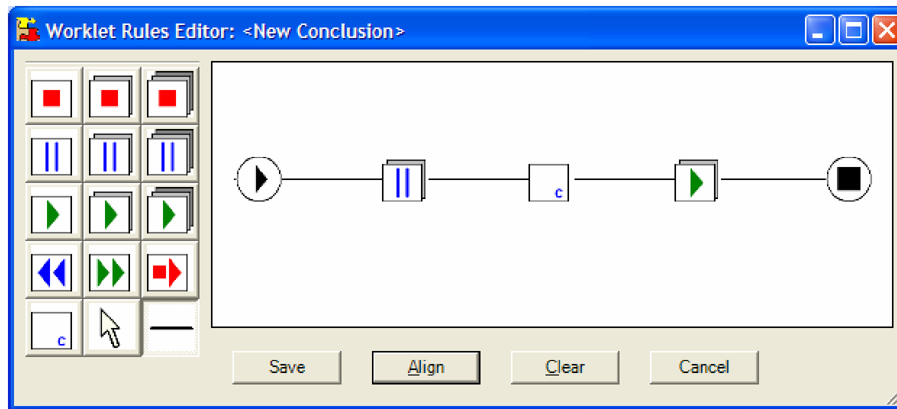

Figure 6.20: Example Handler Process in the Rules Editor

- ResourceUnavailable - occurs when a resource reports that it is unable to accept the allocation of a workitem; and

- ConstraintViolation - occurs on a data constraint violation for a workitem *during* its execution (as opposed to pre- or post- execution).

The exception service maintains a set of ripple-down rules that is used to determine which exlet, if any, to invoke. If there are no rules defined for a certain exception type in the rule set 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.

An example of a definition of an exlet in the Rules Editor can be seen in Figure 6.20 (see Section 6.9 for more details regarding the Rules Editor and its use). On the left of the graphical editor is the set of primitives that may be used to construct an exlet. The available primitives (reading left-to-right, top-to-bottom) are[3]:

- *Remove* WorkItem, Case, AllCases;

- *Suspend* WorkItem, Case, AllCases;

- *Continue* WorkItem, Case, AllCases;

- *Restart WorkItem*;

- *Force Complete WorkItem*;

[3] see Section 4.4.2 for a description of each primitive and its effects.

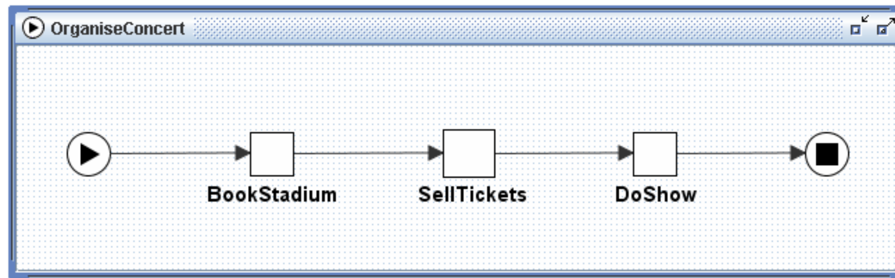

Figure 6.21: The Organise Concert Example Process

- *Force Fail WorkItem*; and
- *Compensate*.

To demonstrate a few of the major features of the exception service, this section introduces the *OrganiseConcert* specification, which is a very simple process modelling the planning and execution of a rock concert. Figure 6.21 shows the specification as it appears in the YAWL Editor.

6.6.1 Constraints Example

As soon as the YAWL Engine launches the case, it notifies the exception service of a new case instance via a PreCaseConstraint event. If the rule set for OrganiseConcert contained a rule tree for pre-case constraints, that tree will be queried using the initial case data to determine whether there are any pre-constraints not met by the case. In this example, there are no pre-case constraint rules defined, so the notification is simply ignored.

Pre-case constraints can be used, amongst other things, to ensure case data is valid or within certain ranges before the case proceeds; can be used to run compensatory worklets to correct any invalid data; or may even be used to cancel the case as soon as it starts (in certain circumstances, if required).

Directly following the pre-case event, the engine notifies the service of a PreItemConstraint for the first workitem in the case instance (in this example, the *Book Stadium* workitem — which does not have a PreItemConstraint rule tree defined for it). The pre-item constraint event occurs immediately when the workitem becomes enabled (i.e. ready to be checked out or executed). Like pre-case constraint rules, pre-item rules can be used to ensure workitems have valid data before they are executed. With each constraint notification, the entire set of case data is made

```
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
```

Figure 6.22: Effective Composite Rule for Do Show's Pre-Item Constraint Tree

available to the exception service — thus the values of any case variables may be queried in the ripple-down rule tree sets for any exception type rule. When the workitem is completed, a PostItemConstraint event is generated by the engine.

The final workitem in the process, *Do Show*, does have a pre-item constraint rule tree, and so when it becomes enabled, the rule tree is queried (see Figure 6.31). The effective composite rule for Do Show's pre-item tree (as viewed in the rules editor), is shown in Figure 6.22.

The rule tree specifies that when *Do Show* is enabled and the value of the case data attribute "TicketsSold" is less than 75% of the seating capacity of the venue, the workitem should be suspended, then the compensatory worklet *ChangeToMidVenue* executed, and then, once the worklet has completed, the workitem should be continued (or unsuspended). Following the flow of the ripple-down rule, if the tickets sold are also less than 50% of the capacity, then after suspending the workitem, the *ChangeToSmallVenue* worklet is to be executed, and then the workitem unsuspended. Finally, if there has been less than 20% of the tickets sold, instead the entire case should immediately be suspended, a worklet run to take the necessary compensatory action to cancel the show, and then finally remove (i.e. cancel) the case.

In the event that the first rule node's condition evaluates to true, for example where a 'Tickets Sold' value is 12600 and the 'Seating' capacity is 25000, the child rule node on the true branch of the parent is tested. Since this node's condition evaluates to false for the example case data, and it has no false child node, the rule evaluation is complete and the last satisfied node's conclusion is returned as the result of the rule traversal.

The result of this process can be seen in the *Available Work* screen of the worklist (Figure 6.23). The *Do Show* workitem is marked with a status of "Suspended" and thus is unable to be selected for checking out; while the *ChangeToMidVenue* worklet has been launched and its first workitem, *Cancel Stadium*, is enabled and may be checked out. By viewing the log file (shown in Appendix C), it can be seen that the *ChangeToMidVenue* worklet is being treated by the exception service as just another case, and so the service receives notifications from the engine for pre-case and pre-item constraint events of the worklet also.

Available Work Items

	ID	Task Description	Status	Enablement Time
○	100005:CancelStadium_3	CancelStadium	Enabled	Mar:26 22:26:00
	100004:DoShow_4	DoShow	Suspended	Mar:26 22:25:59

Check Out   Reset   Raise Exception

Figure 6.23: Work items listed after exlet invocation for Organise Concert

When the last workitem of the worklet has completed, the engine completes the case and notifies the exception service of the completion, at which time the service completes the third and final part of the exlet, i.e. continuing (unsuspending) the *Do Show* workitem so that the parent case can continue. At the *Available Work* screen, the *Do Show* workitem will now show as enabled and thus will be able to be checked out, with the relevant data values entered in the worklet's workitems mapped back to the variables of *Do Show*.

6.6.2 External Trigger Example

It has been stated previously that almost every case instance involves some deviation from the standard process model. Sometimes, events occur completely removed from the actual process model itself, but affect the way the process instance proceeds. Typically, these kinds of events are handled 'off-system' so there is no record of them, or the way they were handled, kept for future executions of the process specification.

The exception service allows for such events to be handled on-system by providing a means for exceptions to be raised by users externally to the process itself. The *Organise Concert* specification will again be used to illustrate how an external exception can be triggered by a user.

A case-level external exception can be raised via the extensions added to YAWL's worklist handler. Figure 6.24 shows the worklist's *Workflow Specifications* screen, and indicates that an instance of the Organise Concert specification is currently executing, since a case identifier is displayed for it (arrowed). Selecting the case identifier displays the Case Viewer screen, which displays four buttons accessible to the user (Figure 6.25). The lower two buttons are worklist extensions provided by the exception service; those buttons appear only when the exception

Active YAWL Specifications

	Specification ID	Spec Name	Documentation	XML	Cases
○	TreatFever	Treat Fever	Worklet to treat a fever	View TreatFever	
○	ChangeToMidVenue	ChangeToMidVenue	Action taken if ticket sales less than expected	View ChangeToMidVenue	
○	Casualty_Treatment	Casualty Treatment	A simple medical treatment process designed to test and demonstrate the Worklet Dynamic Process Selection Service within the YAWL engine	View Casualty_Treatment	
○	OrganiseConcert	Organise Concert	Example used to text workletService exception handling	View OrganiseConcert	22,

Launch Case Reset

Figure 6.24: Workflow Specifications Screen, OrganiseConcert case running

Case Viewer for 22

CaseID : 22
SpecificationID : OrganiseConcert
View State Cancel Case
Raise Exception Reject Worklet

Figure 6.25: Case Viewer Screen for a running OrganiseConcert case (detail)

service is configured as 'enabled'.

When the 'Raise Exception' button is clicked, the *Raise Case Level Exception* screen is displayed (Figure 6.12). This screen is a member of the worklet service's java servlet pages (jsp) (notice the worklet logo in the top left of the banner). Before this screen is displayed, it directly calls a method of the exception service over HTTP and retrieves from the rule set for the selected case the list of existing external exception triggers (if any) for the case's specification. Again, see Figure 6.12 for the list of case-level external triggers defined for the Organise Concert specification.

This list contains all of the external exception 'triggers' that were 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. Notice that at the bottom of the list, the option to add a New External Exception is provided — that option is explained in detail in Section 6.9.3. For this example, we will assume the band has requested some refreshments for backstage. Selecting that exception trigger passes it as the trigger value to the specification's CaseExternalException rule tree and the conclusion for that trigger's rule (i.e. the exlet) is invoked by the service as an

Available Work Items

	ID	Task Description	Status	Enablement Time
○	23:Buy_M_and_Ms_5	Buy_M_and_Ms	Enabled	Sep:12 14:00:54
	22:SellTickets_3	SellTickets	Suspended	Sep:12 13:40:04

Check Out | Reset | Raise Exception

Figure 6.26: Available Work Items after External Exception Raised

exception handling process for the current case.

After the selection is made by the user, the user's worklist is immediately redirected to the *Available Work* screen where it can be seen that the parent case has been suspended and the first workitem of the compensatory worklet, *Organise Refreshments*, has been enabled (Figure 6.26). Once the worklet has completed, the parent case is continued (via the final primitive of the exlet).

Item-level external exceptions can be raised from the *Available Work* screen by selecting the relevant workitem via the radio button to its left, then clicking the *Raise Exception* button (which is another extension to the worklist provided by the exception service), which invokes the *Raise Item Level Exception* screen where the procedure is identical to that described for case-level exceptions, except that the item-level external exception triggers, if any, will be displayed.

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 defining of the conditional expressions of the nodes in the appropriate rule tree or trees that query changes in data values and/or state changes of workitems within the case instance.

6.6.3 Timeout Example

YAWL provides a *Time Service* which, when a workitem is associated with it, will check out the workitem, wait until a pre-defined time span has elapsed or a certain date/time reached, then check the item back in. Effectively, the time service allows a workitem to act as a timer on a process branch; typically one or more workitems execute in parallel to the time service workitem, so that the time service workitem acts as a deadline for the tasks parallel to it. If the deadline is reached, a timeout occurs for that item.

When a workitem times out, the engine notifies the exception service and provides to it

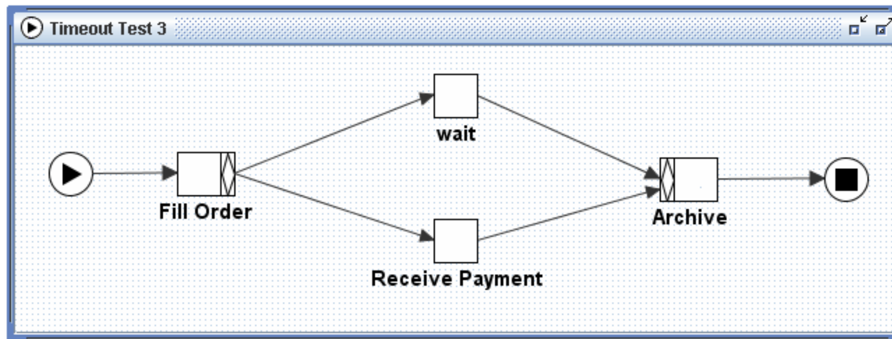

Figure 6.27: The *Timeout Test 3* Specification

a list which includes a reference to all the workitems running in parallel with the timed out workitem, as well as the timed out workitem itself. Thus, rule trees can be defined to handle timeout events for all affected workitems. The specification *Timeout Test 3* provides an example of how a timeout exception may be handled (Figure 6.27).

The first workitem, *Fill Order*, simulates a basic purchase order for a bike. Once the order has been filled, the process waits for payment to be received, before the order is archived. The *wait* task is associated with the Time Service, and so merely waits for some time span to pass. If payment is received before the deadline expires, then the *wait* task is cancelled (because wait is a member of the cancellation region for *Receive Payment*) and the purchase is archived. If the deadline is reached before payment is received, the engine notifies the exception service of the timeout event. The timeout tree set is queried for both the *wait* task and the parallel *Receive Payment* task. For this example, there is a tree defined for the *Receive Payment* task with a single rule (see Figure 6.28).

Notice the rule's condition *isNotCompleted(this)*:

- **isNotCompleted** is an example of a defined function that may be used as (or as part of) a conditional expression in a rule node.
- **this** is a special variable created by the worklet service that refers to the workitem that the rule is defined for and contains, amongst other things, all of the workitem's data attributes and values.

The worklet service provides developers with an easily extendible class where functions can be defined and then used as part of the conditional expressions of rule nodes (see Section 6.8 for more information).

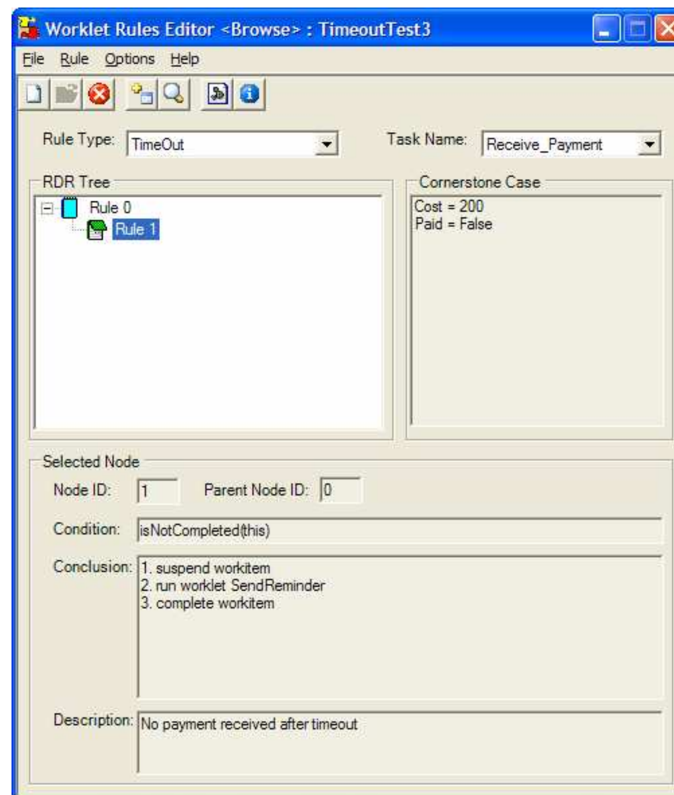

Figure 6.28: Rules Editor Showing Single Timeout Rule for *Receive Payment* Task

In this example, the condition tests if the workitem *Receive Payment* has not yet completed (i.e. if it has a status of *Fired*, *Enabled*, *Executing* or *Suspended*). If it hasn't completed (thus payment for the order has not yet been received) then the conclusion (exlet) will be executed, which includes the launching of the compensatory worklet *SendReminder*. The *SendReminder* worklet consists of three tasks: *Send Request*, and the parallel tasks *wait* and *Receive Reply* - the *wait* task in this specification is again associated with the time service. When its *wait* task times out, the exception service is again notified. The rule set for the SendReminder specification also contains a single timeout rule for the *Receive Reply* task - its condition is again *isNotCompleted(this)* but this time, the rule's conclusion is that shown in Figure 6.29.

When this conclusion's exlet is executed, the *Available Work Screen* in the YAWL worklist now lists workitems from all three cases (Figure 6.30). *File Cancellation* is the first task of

Figure 6.29: Rule detail for *Receive Reply*

Available Work Items

	ID	Task Description	Status	Enablement Time
	30:Receive_Payment_3	Receive_Payment	Suspended	Sep:13 10:48:36
○	32:File_Cancellation_3	File_Cancellation	Enabled	Sep:13 10:49:00
	31:Receive_Reply_5	Receive_Reply	Suspended	Sep:13 10:48:54

Figure 6.30: Available Work After *CancelOrder* Worklet Launched

the *Cancel Order* worklet. Reflected in the *Available Work Screen* is a hierarchy of worklets: case 30 (*TimeoutTest3*) is suspended pending completion of worklet case 31 (*Send Reminder*) which itself is suspended pending completion of worklet case 32 (*Cancel Order*). Thus this example shows that worklets can invoke child worklets to any depth. Notice the third part of the handling process: "remove ancestorCases". Ancestor Cases are all cases from the current worklet case back up the hierarchy to the original parent case that began the exception chain (as opposed to "allCases" which refers to all currently executing cases of the same specification as the case which generates the exception). So, when the *Cancel Order* worklet completes, the *Send Reminder* instance and the original parent *Timeout Test 3* instance are both cancelled by the exception service as a result of the "remove ancestorCases" primitive.

6.7 Ripple Down Rule Sets

A process specification may contain a number of tasks, one or more of which may be associated with the worklet selection service. For each specification that contains a worklet-enabled task, the worklet service maintains a corresponding set of ripple down rules that determine which worklet will be selected as a substitute for the task at runtime, based on the current case data of that particular instance. Each worklet-enabled task in a specification has its own discrete rule tree.

Further, one or more exlets may be defined for a specification and associated with the worklet exception service. A repertoire of exlets may be formed for each exception type. Each specification has a unique rule set (if any), which may contain between one and eight tree sets (or sets of rule trees), one for selection rules (used by the Selection sub-service) and one for each of the seven implemented exception types. Three of those seven relate to case-level exceptions (i.e. CasePreConstraint, CasePostConstraint and CaseExternalTrigger) and so each of

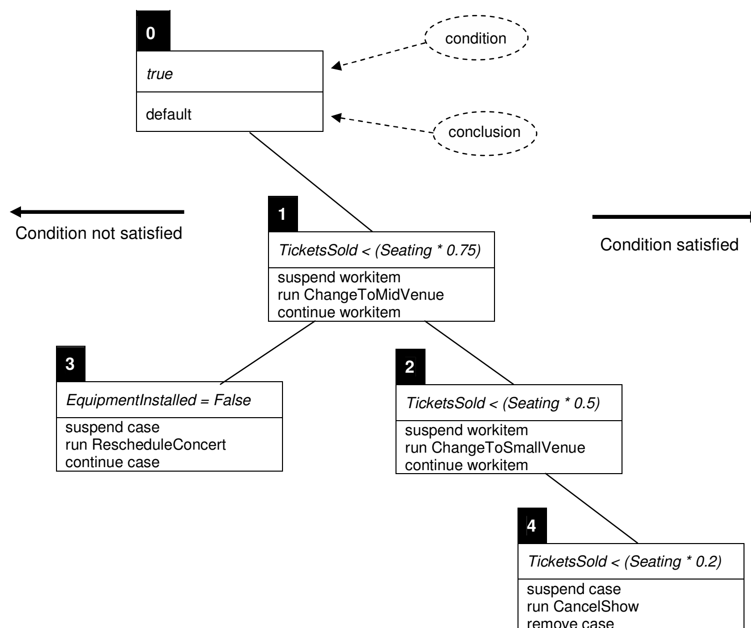

Figure 6.31: Example rule tree (ItemPreConstraint tree for DoShow task of OrganiseConcert)

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.

Thus, 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 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 one

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.

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 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 6.31 shows the ItemPreConstraint rule tree for the Organise Concert example specification (cf. Section 6.6.1), which represents the rule tree for the exlets shown on the centre tier of Figure 6.4.

6.8 Extending the Available Conditionals

The worklet service provides a discrete class that enables developers to extend the set of functions available to be used as part of the conditional expressions of rule sets. The class is called *RdrConditionFunction*, the source code for which currently contains a small number of examples to give developers an indication of the kinds of things that can be achieved with the class and how to create additional functions. The class is freely extensible into any domain space, and thus data can be drawn from diverse sources, such as process mining, resource logs and external databases, and queried through user-defined functions as part of the evaluation of the conditional expressions of rule nodes. The ConditionEvaluator class has been designed to call the RdrConditionFunction object to resolve function references in conditional expressions to their evaluated values. Since the interface between this class and the rest of the service is very small, it can easily be extended and separately recompiled, avoiding the overhead of rebuilding the entire service.

The class code is split into four sections:

- **Header**: containing a listing of available functions and a method to test for the existence of a function;
- **Execute Method**: a pseudo-interface method called by the ConditionEvaluator object

when evaluating a condition containing a user-defined function;

- **Function Definitions**: the functions themselves; and
- **Implementation**: where additional support methods may be created.

To successfully add a function to the class, the following steps must be taken:

1. Create the function (i.e. a normal java method definition) and add it to the 'function definitions' section of the class. Ensure the function:
   - is declared with the access modifier keywords *private static*; and
   - returns a value of *String* type.
2. Add the function's name to the array of *_functionNames* in the header section of the class.
3. Add a mapping for the function in the **execute** method.

Once the function is added, it can be used in any rule's conditional expression.

One example method of the class is called *isNotCompleted*, which is designed to return a boolean true value if a workitem does not have a completed status. The following will use that example to discuss features of the approach.

The first thing required is the addition of the actual function code in the function definition section. Here's the entire *isNotCompleted* function:

```
private static String isNotCompleted(String itemInfo) {
    Element eItem = JDOMConversionTools.stringToElement(itemInfo);
    String status = eItem.getChildText("status");
    return String.valueOf(! isFinishedStatus(status) );
}
```

Notice that the function has been declared as private static and both the argument passed and the returned value is a String type (the interface between the service and this class requires all data to be passed as String types). The first line of the method uses a utility class to convert the string passed into a JDOM Element object. The second line reads the value of the 'status' child Element, while the third line calls the method *isFinishedStatus* (described below).

Note the argument passed to the method (*itemInfo*). When a condition is evaluated by the worklet service for a rule tree owned by a workitem (i.e. any item-level rule), a special constant

called *this* is made available for use in conditional expressions. The variable contains a combination of the properties of the workitem itself (e.g. name, case id, specification id, status and so on) and its data parameters, constructed as a hierarchical JDOM element, then converted to a String representation. When a case-level rule tree is being evaluated, the *this* constant contains the case-level data parameters for the instance invoking the rule.

In this example, the current value of the status of the workitem is read and then passed to the method *isFinishedStatus*, which is defined in the implementation section:

```
/** returns true if the status passed is one of the completed statuses */
private static boolean isFinishedStatus(String status) {
    return status.equals(YWorkItem.Status.Complete) ||
           status.equals(YWorkItem.Status.ForcedComplete) ||
           status.equals(YWorkItem.Status.Failed) ;
}
```

All methods defined in the implementation section must also be declared as private static methods, and may return any data type, but methods in the 'function definition' section must return a String type to conform with the requirements of the interface.

Once the method definition is completed, the name of the function, *isNotCompleted*, has to be added as a String value to the *_functionNames* array in the header section of the code:

```
// add the name of each defined function here
public static final String[] _functionNames = { "max",
                                                "min",
                                                "isNotCompleted",
                                                "today"} ;
```

Finally, the function name must be mapped to the *execute* method, which acts as the interface between the class's functions and the Worklet Service. The execute method receives as arguments the name of the function to execute, and a HashMap containing the function's parameters (again, all are passed as String values). The execute method is essentially an *if . . . else if* block, the sub-blocks of which call the actual functions defined. This is the section of the execute method for the 'isNotCompleted' function:

```
public static String execute(String name, HashMap args) {
    if (name.equalsIgnoreCase("isNotCompleted")) {
        String taskInfo = (String) args.get("this");
        return isNotCompleted(taskInfo);
    }
    else if (name.equalsIgnoreCase("max")) {

    ...
```

```
    }
    return null ;
}
```

The first line checks to see if the name of the function passed to the *execute* method is *isNotCompleted*. If it is, the parameter passed with the function (as String values in the HashMap 'args') is converted to a String value (via the reading of the *this* constant) and finally the *isNotCompleted* function is called — and its return value is passed back from the execute method to the calling ConditionEvaluator object.

The objective of the *RdrConditionFunction* class is to allow developers to easily extend the capabilities of the worklet service by providing the means to test for things in the conditional expressions of rule nodes other than the process instance's data attributes and values. This allows data from any source to be tested during rule evaluations and also provides the service with ease-of-use capabilities when formulating new conditions.

6.9 The Rules Editor

The *Worklet Rules Editor* is a purpose built tool, deployed as part of the worklet repository, that enables the addition of new rules to existing rule sets of specifications, and the creation of new rule sets. It has been developed as a Microsoft .NET based application; the primary reason for choosing a .NET platform was to provide a demonstration of the interoperability of the web- and java- based worklet service with a Windows-based administration tool using HTTP messaging.

The main screen of the editor allows users to graphically view the various parts of each rule node of each rule tree defined for a particular specification. From this screen users can also add new rules to the current rule set, create new tree sets for an existing rule set, and create entirely new rule sets for new specifications. Figure 6.32 shows the main screen with the rule set for the Casualty Treatment specification loaded.

The main features of the screen are explained below.

6.9.1 The Toolbar

The toolbar buttons (Figure 6.33) replicate the functions available from the main menu.

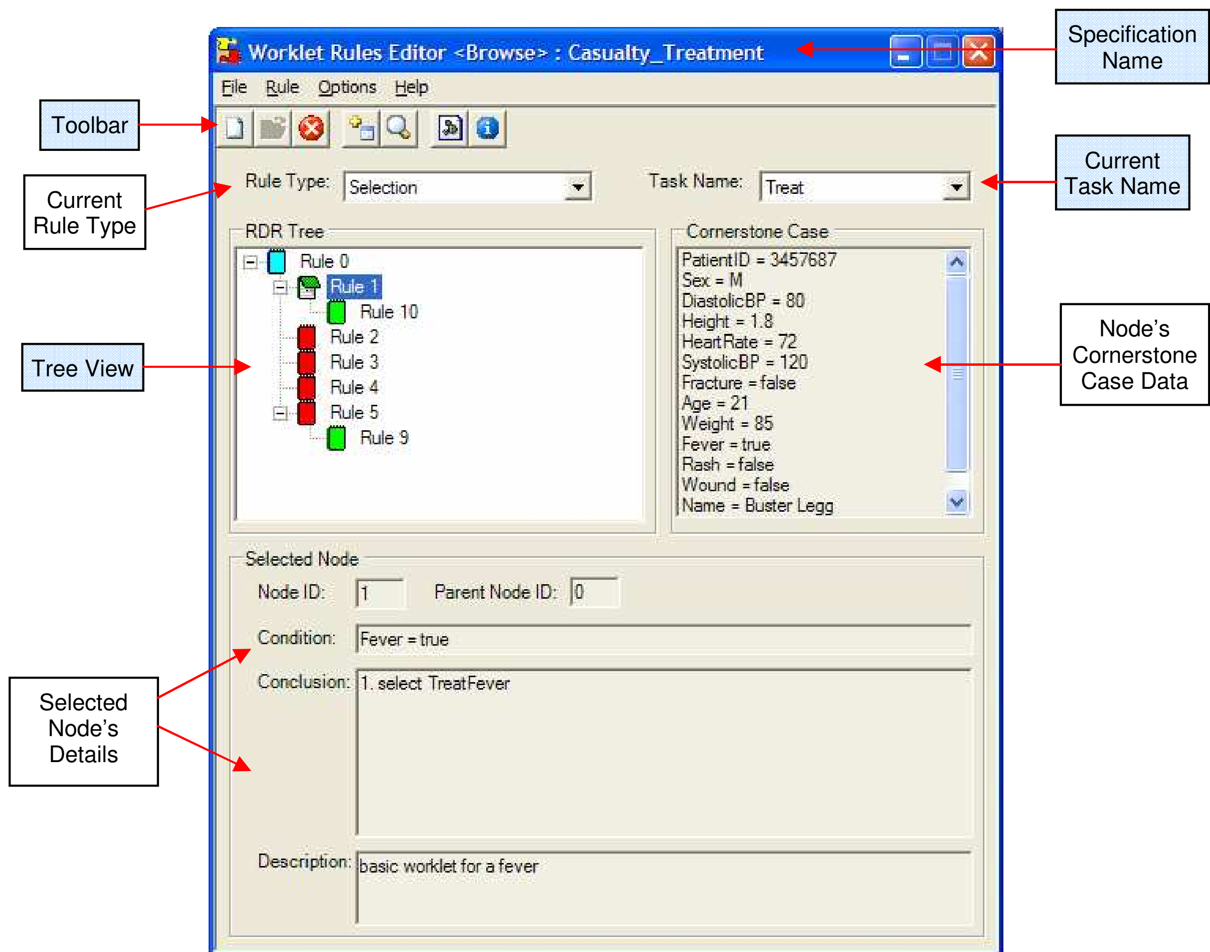

Figure 6.32: Rules Editor Main Screen

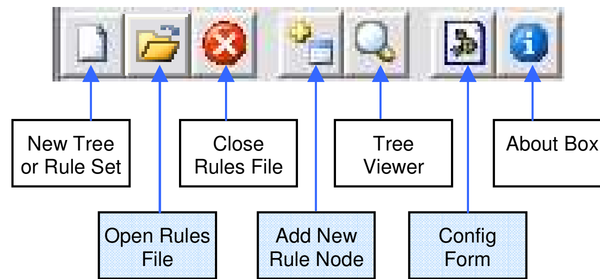

Figure 6.33: The Toolbar

- **New Tree or Rule Set**: If there is no rules file currently open in the Editor, this button displays the *New Rule* form to allow the creation of an entirely new rule set for a specification (i.e. a specification that does not yet have a rule set defined). If there is a rules file currently open in the Editor, the button displays the *New Rule* form to allow the addition of new tree sets to the opened rule set file, but restricted to rule types that have not yet been defined for that specification. See Section 6.9.4 below for more details.

- **Open Rules File**: Opens an existing rules file for browsing and/or editing. The title bar shows the name of the specification associated with the currently loaded rule set.

- **Close Rules File**: Closes the currently opened rules file. Only one rules file may be open at any one time.

- **Add New Rule Node**: Displays the *Add Rule* form to allow the addition of a new rule node to an existing tree to refine a worklet/exlet selection. See Section 6.9.2 below for more details.

- **Tree Viewer**: Displays the *Tree Viewer* form, which provides the ability to view large trees in full-screen mode.

- **Config Form**: Displays the *Configuration* form where paths to the worklet service, repository, specification files and the YAWL editor can be specified.

- **About Box**: Displays some information about the rules editor (version number, date and so on).

- **Other Features**:

  – **Current Rule Type**: This drop-down list displays each rule type that has a tree set defined for it in the opened rules file. Selecting a rule type from the list displays in

the Tree View panel the associated rules tree from the tree set. Works in conjunction with the *Task Name* drop-down list.

- **Current Task Name**: This drop-down list displays the name of each task that has a rules tree associated with it for the rule type selected in the *Rule Type* list. Selecting a task name will display the rules tree for that task in the Tree View. This drop-down list is disabled for case-level rules types.
- **Tree View**: This area displays the currently selected rules tree in a graphical tree structure. Selecting a node in the tree will display the details of that node in the Selected Node and Cornerstone Case panels. Nodes are colour coded for easier identification:
  - A blue node represents the root node of the tree;
  - green nodes are *true* or *exception* nodes (i.e. they are on a true branch from their parent node); and
  - red nodes are *false* or *else* nodes (i.e. they are on a false branch from their parent node).
- **Selected Node**: Displays the details of the node currently selected in the Tree View.
- **Cornerstone Case**: displays the complete set of case data that, in effect, caused the creation of the currently selected rule (see Section 6.9.2 for more details). For example, in Figure 6.32, the Cornerstone Case data shows that, amongst other things, the variable *Fever* has a value of true, while the variables *Rash*, *Wound* and *Fracture* each have value of false.

6.9.2 Adding a New Rule

There are occasions when the worklet launched for a particular case, while the correct choice based on the current rule set, is an inappropriate choice for the case. For example, if a patient in a *Casualty Treatment* case presents with a rash **and** a heart rate of 190, while the current rule set correctly returns the *TreatRash* worklet, it is desirable to treat the racing heart rate before the rash is attended to. In such a case, as the worklet service begins execution of an instance of the *TreatRash* process, it becomes obvious to the user (in this case, a paramedical) that a new rule needs to be added to the rule set so that cases that have such data (both now and in the future) will be handled correctly.

Every time the worklet service selects a worklet or exlet to execute for a specification instance, a log file is created that contains certain descriptive data about the worklet selection process. These files are stored in the worklet repository. The data stored in these files are in XML format, and the files are named according to the following format:

CaseID_SpecificationID_RuleType_WorkItemID.xws

For example: *12_CasualtyTreatment_Selection_Treat.xws* (xws for XML Worklet Selection). The identifiers in the filename refer to the parent specification instance, not the worklet case instance. Also, the WorkItemID identifier part will not appear for case-level rule types. So, to add a new rule after an inappropriate worklet choice, the particular selected log file for the case that was the catalyst for the rule addition must be located and loaded into the rules editor's *Add Rule* form. Before a rule can be added, the appropriate rule set must first be loaded into the editor. Note that the selected file chosen must be for an instance of the specification that matches the specification rule set loaded in the editor (in other words, you can't attempt to add a new rule to a rule set that bears no relation to the xws file opened here). If the specifications don't match, an error message will display.

Figure 6.34 shows the *Add Rule* form with the selected file loaded. The *Cornerstone Case* panel shows the case data that existed for the creation of the original rule that resulted in the selection. The *Current Case* panel shows the case data for the current case — that is, the case that is the catalyst for the addition of the new rule. The *New Rule Node* panel is where the details of the new rule may be added. Notice in Figure 6.34 that the identifiers of the parent node and the new node are shown as read only — the rules editor takes care of where in the rule tree the new rule node is to be attached, and whether it is to be added as a true child or false child node, using the algorithm described in Section 4.2. The parent node of the new rule node is the node that was returned as the 'last searched node' included in the result of the original rule tree evaluation.

Since the case data for the original rule and the case data for the new rule are both displayed, to define a condition for the new rule it is only necessary to determine what it is about the current case that makes it necessary for the new rule to be added. That is, it is only where the values for the corresponding case data attributes differ that distinguishes one case from the other, and further, only a subset of that differing data is relevant to the reason why the original selection was inappropriate. For example, there are many data items that differ between the two case

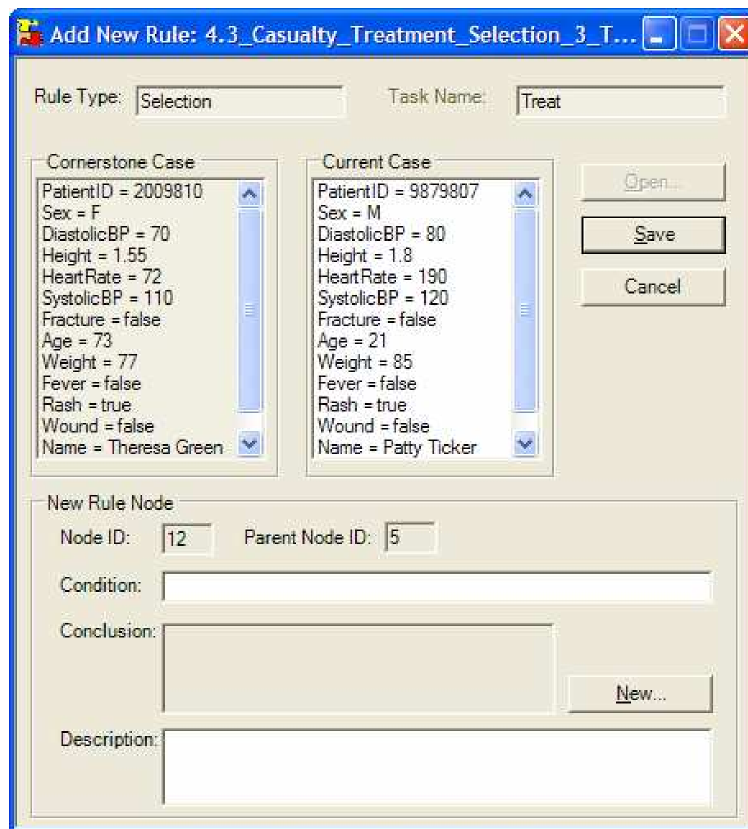

Figure 6.34: Rules Editor (Add New Rule Screen)

data sets shown in Figure 6.34, such as PatientID, Name, Sex, Blood Pressure readings, Height, Weight and Age. However, the only differing data item of relevance here is HeartRate — that is the only data item that, in this case, makes the selection of the *TreatRash* worklet inappropriate.

Clicking on the line "HeartRate = 190" in the *Current Case* panel copies that line to the *Condition* input in the *New Rule Node* panel. Thus, a condition for the new rule has been easily created, based on the differing data attribute and value that has caused the original worklet selection to be invalid for this case.

Note that it is not necessary to define the rule as "Rash = True & HeartRate = 190", as might first be expected, since this new rule will be added to the true branch of the *TreatRash* node. By doing so, it will only be evaluated if the condition of its parent, "Rash = True", first evaluates to *True*. Therefore, any rule nodes added to the true branch of a parent become exception rules of the parent. In other words, this particular complete tree traversal can be interpreted as: "if Rash is True then return TreatRash except if HeartRate is 190 then return ???" (where *???* denotes whatever worklet we decide to return for this rule — see more below).

As it stands, the conditional expression for the new rule would be sufficient if, in future

Table 6.2: Operator Order of Precedence

Precedence	**Operators**	**Type**
1	$*$ $/$	Arithmetic
2	$+$ $-$	
3	$=$ $<$ $<=$	Comparison
	$>$ $>=$ $!=$	
4	&	Logical AND
	$\vert$	Logical OR
	!	Logical NOT

cases, a patient's heart rate will be exactly 190, but what if it is 191, or 189, or 250? Clearly, the rule needs to be amended to capture all cases where the heart rate exceeds a certain limit: say 175. While selecting data items from the *Current Case* panel is fast and easy, it is often the case that the condition needs to be further modified to correctly define the relevant rule.

The *Condition* input of the *Add Rule* form allows direct editing of the inserted condition. Conditions are expressed as strings of operands and operators of any complexity, and sub-expressions may be parenthesised. Table 6.2 shows the supported operators and their order of precedence.

All conditions must finally evaluate to a Boolean value. To make the condition for the new rule more appropriate in this example, the condition “HeartRate = 190” should be edited to read “HeartRate $>$ 175”.

After defining a condition for the new rule, the name of the appropriate worklet or exlet to be executed when this condition evaluates to true must be entered in the *Conclusion* field of the *New Rule Node* panel via the *New* button (refer Figure 6.34). If the new rule is to be added to a selection tree, the process to add the rule is that explained below. Refer to Section 6.9.4 for details on adding a conclusion for the exception rule types.

For a selection rule tree, when the *New* button is clicked, a dialog is displayed that comprises a drop-down list containing the names of all the worklets in the worklet repository (refer Figure 6.35). An appropriate worklet for this rule may be chosen from the list, or, if none of the existing worklets are suitable, a new worklet specification may be created.

Clicking the *New* button on this dialog will open the YAWL Editor so that a new worklet specification can be created. When the new worklet is saved and the YAWL Editor is closed,

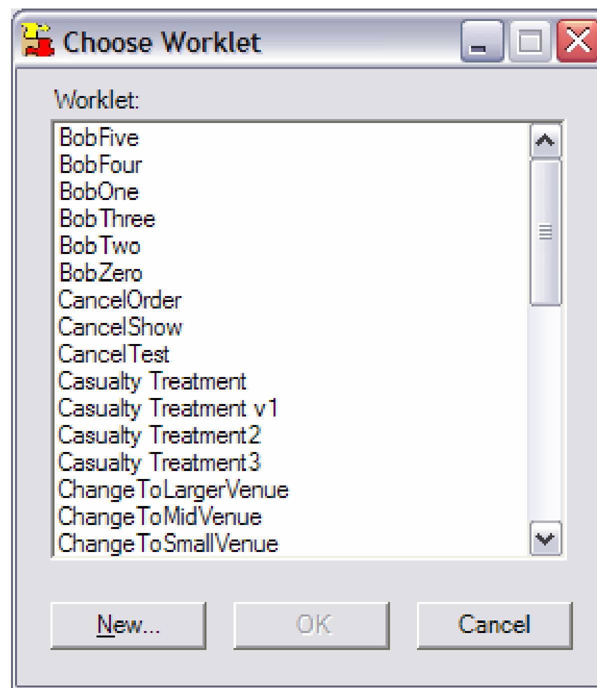

Figure 6.35: The Choose Worklet dialog

the name of the newly created worklet will be displayed and selected in the worklet drop-down list. Once all the fields for the new rule are complete and valid, clicking the *Save* button adds the new rule to the rule tree in the appropriate location.

6.9.3 Dynamic Replacement of an Executing Worklet

The creation of the new rule in the running example above was triggered by the selection and execution of a worklet that was deemed an inappropriate choice for the current case. So, when a new rule is added, administrators are given the choice of replacing the executing (inappropriate) worklet instance with an instance of the worklet defined in the new rule.[4] After saving the new rule, a dialog similar to that in Figure 6.36 is shown, providing the option to replace the executing worklet, using the new rule. The message also lists the specification and case identifier's of the original work item, and the name and case id of the launched worklet instance.

If *Yes* is clicked, then in addition to adding the new rule to the rule set, the rules editor will contact the worklet service using HTTP messaging and request the change. A further message dialog will be shown soon after with the results of the replacement process sent from the service back to the editor, similar to that in Figure 6.37. If the *No* button is clicked, then the new

[4]When a worklet instance is rejected, it is automatically suspended pending replacement.

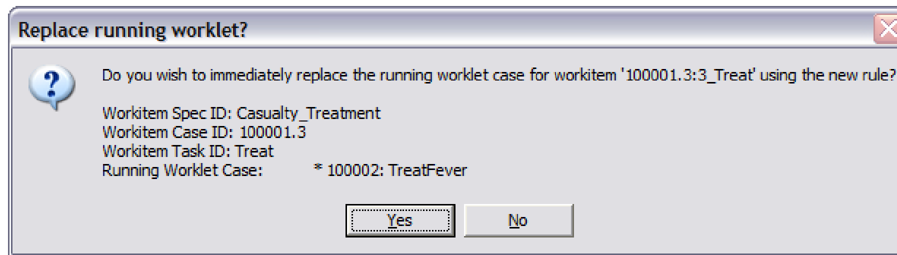

Figure 6.36: Dialog Offering to Replace the Running Worklet

rule is simply added to the rule set, and the original process un-suspended. An administrator would choose to replace the rejected worklet in almost every instance, but the option remains to effectively overrule the worker's worklet rejection depending on organisational rules, authorities and constraints.

6.9.4 Creating a New Rule Set and/or Tree Set

As mentioned previously, it is not necessary to create tree sets for all of the rule types, nor a rule tree for an item-level rule type for each and every task in a specification. So, most typically, working rule sets will have rule trees defined for a few rule types, with other types left without rules trees defined for them (any events that don't have associated rules for that type of event are simply ignored). It follows that there will be occasions where it becomes necessary to add a new tree set to a rule set for a previously undefined rule type, or add a new tree, for a task that has no rule tree for a particular rule type previously defined, to an existing tree set. Also, when a new specification has been created, a corresponding base rule set will also need to be created (if selections and exceptions for the new specification are to be handled by the service).

For each of these situations, the rules editor provides a *Create New Rule Set* form (see Figure 6.38), which allows for the definition of new rule trees (with any number of rule nodes) for existing tree sets (where there is a task of the specification that has not yet had a tree defined

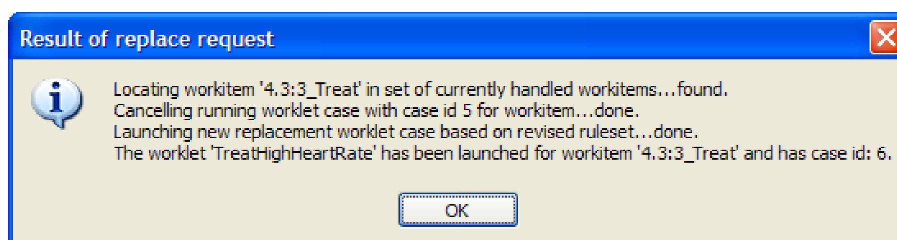

Figure 6.37: Example Dialog Showing a Successful Dynamic Replacement

Figure 6.38: Create New Rule Set Form

for it within the tree set); the definition of new tree sets for specifications that have not yet had a tree set defined for a particular rule type; and entirely new rule sets for specifications that have not yet had a rule set created for them. The form allows administrators to create a rule set, one rule tree at a time (for the selected specification, rule type and, if applicable, task name); its use varies slightly depending on whether it is working with a new rule set or an existing rule set.

This section describes the features of the *Create New Rule Set* form for adding a new rule set, and points out how the process differs for existing rule sets where required. The creation of a new rule set begins by clicking the *New Rule* toolbar button. On the form:

- The *Process Identifiers* panel is where the names of the specification, rule type and, if applicable, task name for the new tree are defined. The *Specification Name* input is read-only — for new rule sets it is the specification chosen via a *Specification Location* dialog displayed when the form is first opened; for existing rule sets it is the specification for the rule set currently loaded into the editor. The *Rule Type* drop-down list contains all of the available rule types (i.e. all the rule types for which no or incomplete tree sets exist). For new rule sets, all rule types are available. The *Task Name* drop-down list contains all

the available tasks for the selected rule type (i.e. tasks for which no tree exists in the tree set for this rule type). The names of all tasks defined for a specification are automatically gathered by the editor's parsing of the particular specification file. The *Task Name* list is disabled for case-level rule types.

- The *New Rule Node* panel is identical to the panel on the *Add New Rule* form. Here a condition and optional description can be entered, and the conclusion for the new rule created or selected from the list (depending on the rule type — see below).
- The *Cornerstone Case Data* panel allows a set of cornerstone data for the new rule to be defined, via the Attribute and Value inputs, and the Add button. The standard XML naming rules apply to the data attributes: the attribute name must begin with an alpha character or underscore and contain no spaces or special characters.
- The *Effective Composite Rule* panel displays a properly indented textual interpretation of the composite condition comprising the conditions of the selected node and all ancestor nodes back to the root node - in other words, the entire composite condition that must evaluate to true for the conclusion of the selected node to be returned.
- The *RDR Tree* panel dynamically and graphically displays the new rule tree as it is being created.

New rule nodes can be added wherever there is a node on the tree that does not have a child node on both its true and false branches (except of course the root node which can have a true branch only). To identify available locations where a new rule node may be added, special 'potential' nodes are displayed in the *RDR Tree* panel, called "New True Node" or "New False Node". These potential nodes are coloured yellow for easy identification. Selecting a yellow new rule node enables the various inputs for the new rule on the form (they are disabled or read-only by default). Clicking the *New* button adds a conclusion for the new rule. If the currently selected rule type is 'Selection', a worklet can be added as a conclusion in the manner described in Section 6.9.2.

If it is one of the exception rule types, the *New* button will display the *Draw Conclusion* dialog, allowing for the graphical creation of an exlet process model comprising a sequence of tasks (or primitives), as explained in detail in Section 6.9.5 below. When the conclusion

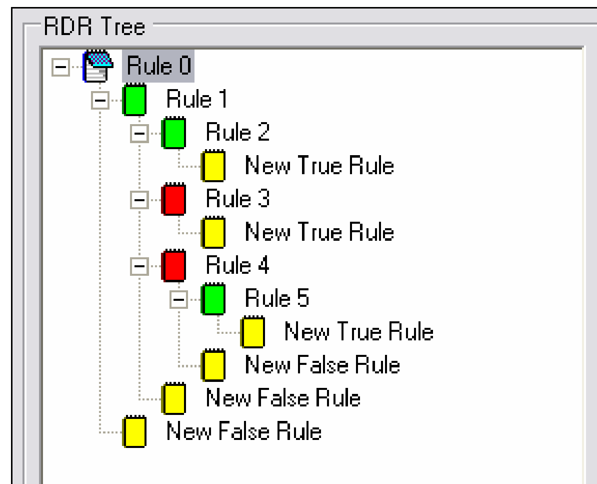

Figure 6.39: Creating a New Rule Tree

sequence has been defined and the dialog closed, a textual representation of it will be displayed in the *Conclusion* panel.

Once the new rule node has been given a condition and conclusion, and optionally some cornerstone data and a description, clicking the *Add Rule* button adds the rule to the tree. The new node will be displayed at the selected position on the tree with the relevant coloured node icon indicating whether it is a true or false node of its parent. New potential node add-points will also be displayed. Figure 6.39 is an example of a newly created tree that has had several nodes added.

The add rule procedure may be repeated for however many rule nodes are to be added by clicking on the appropriate yellow (potential) nodes. When node addition is complete, clicking the *Add Tree* button will add the tree just created to the tree set selected (via the selected Rule Type and, if applicable, Task Name lists).

Once the newly created tree has been added to the selected tree set administrators will no longer be able to add nodes to the tree via the *New Rule Set* form. This is to protect the integrity of the rule set. Since each subsequent rule will be added because of an exceptional case or where the selected worklet does not fit the context of a case instance, the preferred method is to create the base rule set and then add rules as they become necessary via the *Add Rule* form as described earlier. In this way, added rules are based on real case data and so are guaranteed to be valid. In a similar vein, there is no option to modify or delete a rule node within a tree once the tree has been added to the rule set, since to allow it would destroy the integrity of the rule set, because the validity of a child rule node depends on the conditional expressions of its

ancestors.[5]

When a tree is added to the tree set:

- If it is a case-level tree, the rule type that the tree represents will be removed from the *Rule Type* list. That is, the rule type now has a tree defined for it and so is no longer available for selection on the *New Rule* form.
- If it is an item-level tree, the task name that the tree represents will be removed from the *Task Name* list. That is, the task now has a rule tree defined for it (for the selected rule type) and so is no longer available.
- If it is an item-level tree, and all tasks of the specification now have trees defined for them for the selected rule type (i.e. this was the final task of the specification for which a tree was defined), the rule type that the tree represents will be removed from the *Rule Type* list.

This approach ensures that rule trees can only be added where there are currently no trees defined for the selected specification. Once the tree is added, the form resets to allow the addition of another new tree as required, by repeating the process above for a new rule type (or rule type/task name for item-level trees).

6.9.5 Drawing a Conclusion Sequence

As described in Section 6.9.2, adding a conclusion to a selection rule is a matter of choosing a worklet from the list or creating a new worklet in the YAWL Editor. However, when adding a conclusion for a rule type other than 'Selection' (i.e. an exception rule type), an exlet needs to be defined that will be invoked when the rule is returned as the last satisfied. Section 6.6 detailed the various actions that make up the available set of exception handling primitives (or tasks) that may be sequenced to form an entire handling process. The *Draw Conclusion* dialog makes the process of defining an exception handling sequence easier by allowing administrators to create an exlet graphically by selecting the appropriate primitive from the toolbox on the left, and then clicking on the drawing canvas to place the selected primitive. Figure 6.40 shows an example of the *Draw Conclusion* dialog.

[5] There are algorithms defined in the literature for the reduction and transformation of ripple-down rule sets that may be applied from time to time (for example, see [156]).

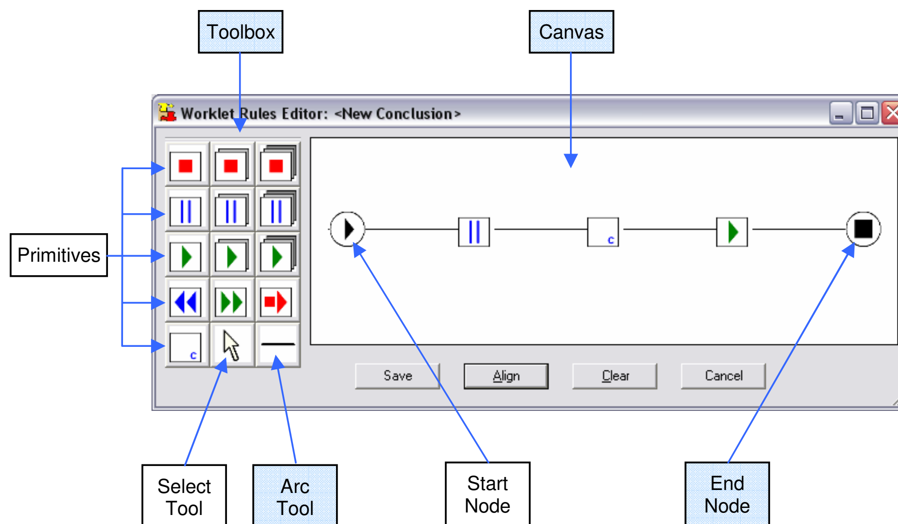

Figure 6.40: The Draw Conclusion Dialog

In addition to the various primitive tools, the *Arc Tool* is used to define the sequence order. For a conclusion to be valid (and thus permitted to be saved) there must be a direct, unbroken path from the start node to the end node (the start and end nodes are always displayed on the drawing canvas). Also, the conclusion will be considered invalid if there are any nodes on the canvas that are not attached to the sequence when a save is attempted.

The *Select Tool* is used to move placed primitives around the canvas. The *Align* button will immediately align the nodes horizontally and equidistantly between the start and end nodes (as in Figure 6.40). The *Clear* button will remove all added nodes to allow a restart of the drawing process. The *Cancel* button discards all work and returns to the previous form. The *Save* button will save the conclusion and return to the previous form (as long as the exlet is deemed valid).

The *Compensate* primitive will, when invoked at runtime, execute a worklet as a compensation process as part of the handling exlet process. To specify which worklet to run for a selected compensate primitive, a popup menu is provided which invokes the *Choose Worklet* dialog (identical to the dialog shown for a ‘Selection’ conclusion process) allowing the selection of an existing worklet or the definition of a new worklet to run as a compensatory process. Selecting the appropriate worklet adds it to the compensation primitive. An exlet will be considered invalid (and thus unable to be saved) if it contains a compensate primitive for which a worklet has not yet been defined.

The primitives *SuspendAllCases*, *RemoveAllCases* and *ContinueAllCases* may be optionally limited to ancestor cases only via the popup menu associated with those kinds of primitives. An-

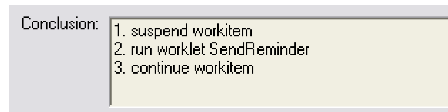

Figure 6.41: A Conclusion Sequence shown as Text (detail)

cestor hierarchies occur where a worklet is invoked for a case, which in turn invokes a worklet, and so on. When a primitive is limited to ancestor cases, it applies the primitive's action to all cases in the hierarchy from the current case back to the original parent case, rather than all running cases of the specification.

When a valid exlet is saved, the editor returns to the previous form (i.e. either the *Add Rule* or *New Rule* form depending on from where it was invoked). The conclusion will be displayed textually as a sequential list of action-target pairs (an example can be seen in Figure 6.41).

6.9.6 Rules Editor Internal Structure

This section offers an overview of the structure of the Rules Editor tool. As mentioned earlier, the editor is built on the .NET Framework and is written using the Visual Basic .NET programming language. Figure 6.42 shows the seventeen classes that make up the application.

The top nine classes represent the *forms* of the application. When using VB .NET, each screen in an application consists of a 'form', and is a class encapsulating both the window, GUI controls and graphical environment, and the additional attributes and methods that provide the required functionality particular to that form. It can be seen in Figure 6.42 that the form classes correspond to those forms discussed in the earlier sections describing the functionality of the editor. Of particular interest is the *frmDrawConc* class, which could be described as a complete application within an application. The form provides the capabilities to allow administrators to construct exlets graphically. Its structure is displayed in Figure 6.43.

The *ProcNode* internal class stores the details of each primitive in the exlet being constructed; thus the exlet is stored internally as a doubly-linked list. Each arc in the exlet process is represented by an *sArc* structure, which has fields that describe the *ProcNodes* at either end of the arc, as well as the screen coordinates of the arc itself.

The *frmDrawConc* class contains methods for adding and deleting primitives and arcs (e.g. AddNode, RemoveNode, DrawArc, RefreshLineAttachPoint), aligning nodes and arcs (e.g. ReAlignNodes, RedrawArcs), validating the graph (e.g. hasCompleteCompensations, HasNoLooseN-

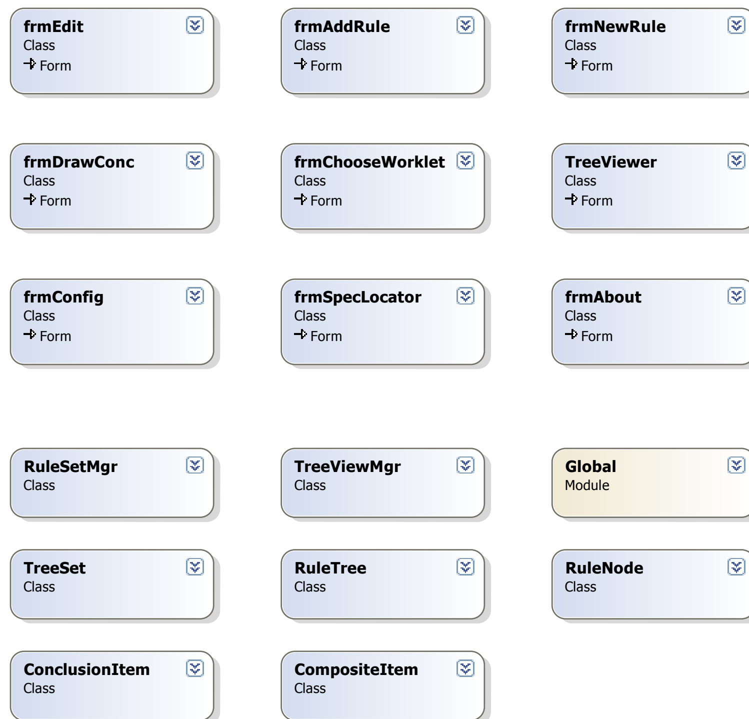

Figure 6.42: Rules Editor Classes

odes, isCompleteGraph) and translating between graphical and textual representations of exlets (ComposeGraphXML, ConclusionItemToToolType, ShowGraph).

The two 'manager' classes, *RuleSetMgr* and *TreeViewMgr*, act as bridging classes between various forms and their underlying functionality. *TreeViewMgr* manages the consistent graphical display of rule trees on the *frmEdit*, *frmNewRule* and *TreeViewer* forms. *RuleSetMgr* is the heart of the application, managing the construction and manipulation of rule nodes, trees and sets. Figure 6.44 shows the structure of the *RuleSetMgr* class.

This class keeps track of the rule set (that is, the set of tree sets) for a specification. It contains methods for reading and writing rule sets to and from their XML files for storage, adding and updating trees and tree sets, and transforming various elements of tree sets into other formats as required by the forms *frmEdit*, *frmNewRule*, *frmAddRule* and *frmDrawConc*. The field *_ruleset* stores the tree sets for the loaded specification. Each rule set may have up to eleven tree sets, one for each exception type. Each tree set may consist of a number of rule trees, each with a number of rule nodes. The conclusion of each rule node is made up of an array of *ConclusionItem* objects (one for each primitive in the exlet that represents the conclusion), and the cornerstone data of each node is made up of an array of *CompositeItem* objects (attribute-value pairs). Figure 6.45 shows the relationship between these classes.

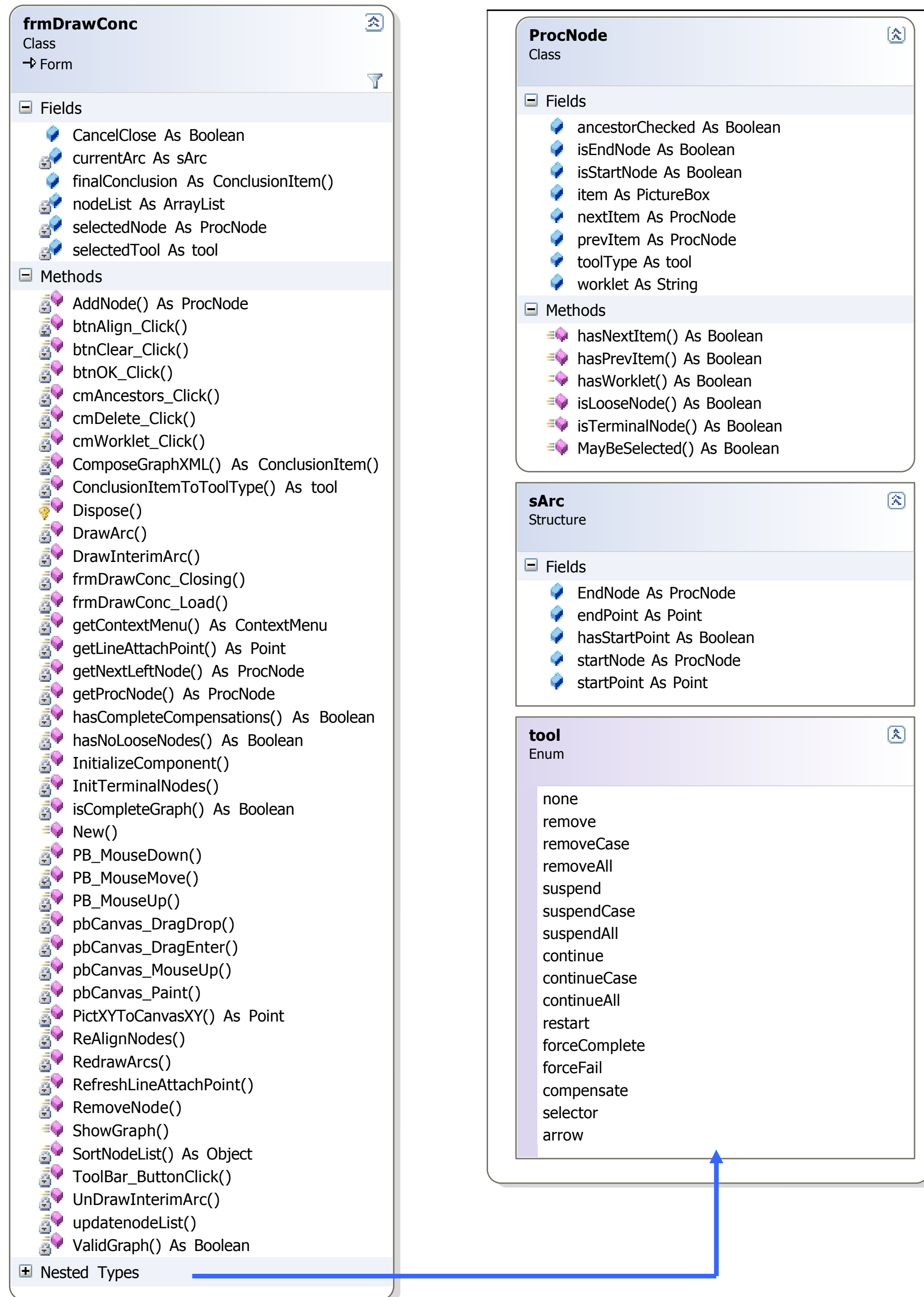

Figure 6.43: The frmDrawConc class

RuleSetMgr
Class

Fields
- _fName As String
- _ruleSet As TreeSet()
- _spec As String
- _specFullPath As String
- _taskNames As String()

Properties
- hasFileLoaded As Boolean
- item As TreeSet
- LoadedFileName As String
- SpecFullPath As String
- SpecName As String
- TaskNames As String()

Methods
- addTreeToRuleSet()
- ExtractSpecName() As String (+ 1 overload)
- getAvailableTaskNamesForTreeType() As String()
- GetLoadedRuleTypesAsStrings() As String()
- GetLoadedTasksForRuleType() As String()
- getRuleTypeList() As String()
- getTaskListFromSpec() As String()
- IntToTreeType() As exType
- isCaseLevelTree() As Boolean (+ 1 overload)
- LoadConstraintRules()
- LoadExternalRules()
- LoadItemRules() As TreeSet
- LoadRulesFromFile() As Boolean
- LoadTree() As RuleTree
- loadVersionOneRules() As TreeSet
- New() (+ 1 overload)
- ReadCompositeItem() As CompositeItem()
- ReadConclusion() As ConclusionItem()
- ReadRuleNode() As RuleNode
- ReadTaskNames() As ArrayList
- SaveRulesToFile()
- setTreeSetTags()
- StringToTreeType() As exType
- TreeNameToType() As exType
- TreeTypeToString() As String
- updateTreeInRuleSet()
- WriteElement()

Nested Types

exType
Enum

- CasePreConstraint
- CasePostConstraint
- ItemPreConstraint
- ItemPostConstraint
- ItemAbort
- TimeOut
- ResourceUnavailable
- ConstraintViolation
- CaseExternalTrigger
- ItemExternalTrigger
- Selection

Figure 6.44: The RuleSetMgr class

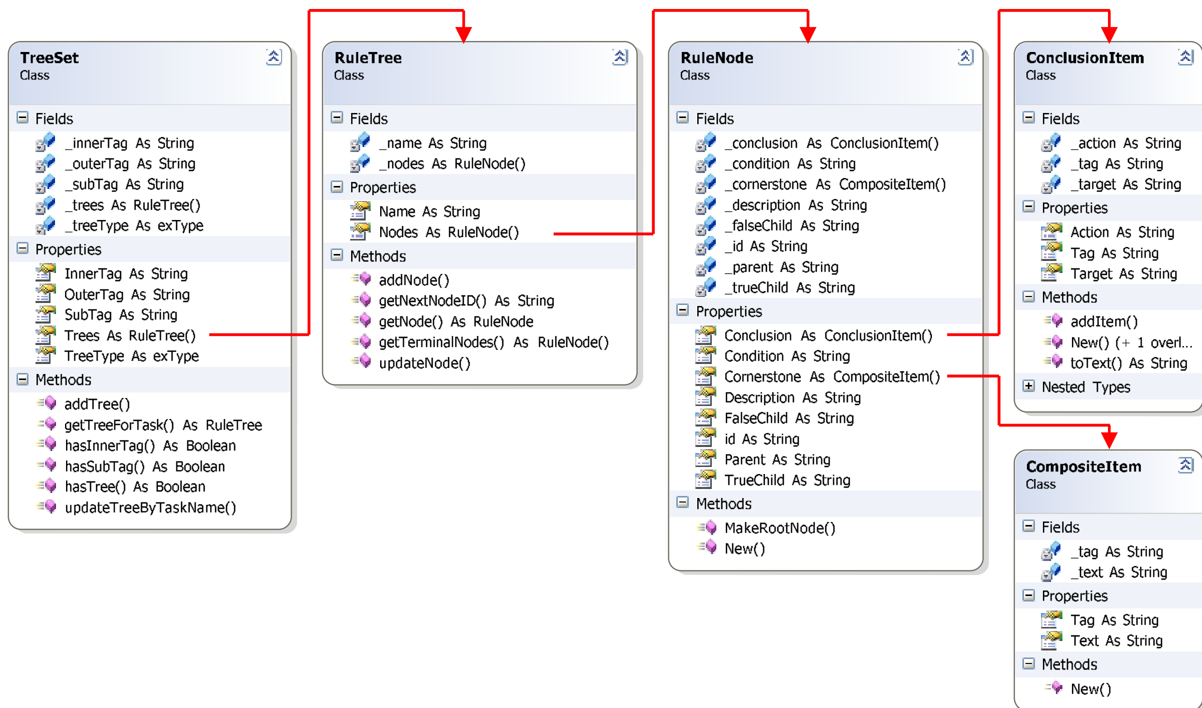

Figure 6.45: The Rule Classes

Conclusion

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 provides innovative techniques that 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 flexibility and 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).

Chapter 7

Validation

In order to validate the viability of, and benefits offered by, the worklet approach, two exemplary studies have been undertaken: one involving a relatively rigid business scenario and the other a more creative environment. Each study compares current methods of the management and operation of workflow processes in the selected organisation with the way they could be managed using a worklet service approach.

The first study involves a process of an organisation that provides telecommunications and energy services in the United Kingdom. It provides an interesting study for this research because the organisation (i) currently uses a YAWL environment to support the workflow of several processes involving Customer Relationship Management (CRM); and (ii) has identified the potential for the occurrence of exceptions at various points during the execution of process instances and has taken steps to augment the static YAWL environment to accommodate an approach to handling them. A selected business case is presented, and the organisation's approach discussed before an examination of a worklet solution is given.

The second study examines two processes performed in a very creative environment: a Film Production company. In this study, the organisation operates in a highly ad-hoc, goal-oriented manner and consequently their processes are not currently managed by a workflow system. In fact, team members were somewhat sceptical of the benefits that could be offered by a workflow management solution. The processes are first described using an C-EPC format, before a worklet solution to each is provided.

In each case, the advantages of the worklet paradigm over current methods will be made apparent.

7.1 Exemplary Study - Business Environment

7.1.1 Background

First Utility Ltd (first:utility) is an independent utilities group established in 1994 and based in the United Kingdom. Amongst other interests, it provides a variety of competitive telephony services and telecommunications packages to their customers, offering an alternative to British Telecom.

The organisation leases wholesale telephony lines from British Telecom (BT) (who operate the main domestic residential telephony networks in the United Kingdom); the product offered by BT is known as *Wholesale Line Rental* (WLR). first:utility then on-sell the lines to their customers as an individualised product that offers a single monthly billing structure for calls and line rental at a competitive rate.

British Telecom provide an online system called the *Service Provider Gateway* (SPG) which provides access for service providers like first:utility to components of BT's internal systems. In particular, the SPG can be used to place a "WLR Transfer" order, which initiates the transfer of a customer's telephone from BT to a service provider, and invokes a process for the necessary network reconfiguration at the local telephone exchange. The SPG also provides features to track the progress of individual orders, to cancel a current order and to obtain summary reports of previously placed orders.

The SPG can be accessed via both an interactive HTTPS based browser session or via an 'XML over HTTPS' interface. Because the browser based interface is open to both human input and transaction errors, it is not suitable for high volumes of order processing. Therefore, the system developed by first:utility to interact with the SPG utilises the automated XML over HTTPS interface such that all interaction with the SPG can be controlled and arbitrated via automated systems. An implementation of the YAWL environment is used to provide workflow support for both the back-office automation and human-controlled work activities of their Customer Relationship Management (CRM) systems.

It became evident at an early stage that interaction with the SPG and processing of WLR Transfer orders involves the potential for a number of exceptions to occur and, as a consequence, a method for handling exceptions, including suitable recovery and retry processing, was essential.

A number of exceptions, corresponding to various levels of Eder & Liebhart's hierarchy [67], were identified, such as:

- **Basic failures**: for example physical outage of the SPG server system, SSL handshake failures over the HTTPS network, and X509 certificate lifetime expiry;
- **Application failures**: such as rejections resulting from malformed XML in order documents;
- **Expected exceptions**: including SPG authentication password expiries, 'random' rejections resulting from BT's records concerning the telephone line being transferred, order cancellations and other processing failures, which as a group involve deviations from the 'normal' or expected flow of the process; and
- **Unexpected exceptions**: while not directly catered for within workflow enactments, any unexpected exceptions are detected, logged and manually progressed such that the offending operation can be corrected and retried, transparent to the YAWL runtime.

The following sections discuss the first:utility approach, and then compare it to the worklet service solution, to show the additional features and enhanced exception handling techniques that are made available by the worklet service.

7.1.2 The first:utility Approach

The approach used by first:utility to handle process-level exceptions raised by applications performing a task involves storing an error response code from the application into a variable at the task decomposition level and then mapping that to a corresponding net-level variable. The value of that variable is then tested via a conditional branch explicitly modelled in the process control flow (i.e. via an OR or XOR split), and the appropriate branch taken. This approach requires all possible branches, that is control-flow branches for both the 'normal' process and those dealing with exceptions, to be incorporated into the static process model.

An example of such an (exception) branch is one which handles a password expiry at the SPG interface. The return code from the interface is tested at an XOR split construct in the process model, and in this scenario control is routed to a task which appears in a user's worklist requesting that the password be reset. Once reset, the branch loops back to the original task for

a retry of the interface using the new password. Thus a looping branch has been added to the business logic of the process model to accommodate one potential expected exception.

A more detailed example is the *FixedLineWLRWithdraw* specification model, shown in Figure 7.1, that coordinates the processing of a request from the front-office CRM system to withdraw an existing WLR service from a customer's account following a termination request. The 'normal' or expected control flow (i.e. the business logic of the process) is as follows:

- the customer's audit record is updated to record the withdrawal request;
- an order to process the withdrawal of the service is placed on the SPG;
- the service is provisionally flagged as being withdrawn in the company database;
- the customer's audit record is updated to indicate acknowledgement by the SPG of the request;
- the status of the request is monitored via the SPG at 24 hour intervals;
- once the order has 'completed' status, the service withdrawal is committed to the database; and
- the customer's audit record is updated a final time to signal success.

This process model shows two XOR splits (on the *PLACEORDER* and *QUERYORDER* tasks) where specific routing of process control flow occurs depending on whether the output values returned from the SPG to those tasks contain an error-code value. Thus two of the three main control flow branches exist in this model to accommodate exception handling. Additionally, there is a looping construct around the *QUERYORDER* task which routes to the *wait24H* task (associated with the YAWL time service), which basically serves as a retry in the event the original order query does not return a completed status notification.

The atomic task *PLACEORDER* is performed by a YAWL Custom Service developed by first:utility that passes a request to withdraw a telephone service to the SPG interface for processing. Details describing the specific telephone service to withdraw are captured via a separate process and loaded into the *FixedLineWLRWithdraw* instance when it is initiated. If the request completes successfully, the process flow routes to the *removeServiceProv* atomic task, which removes the instance of the WLR service from the customer's database record. The branch

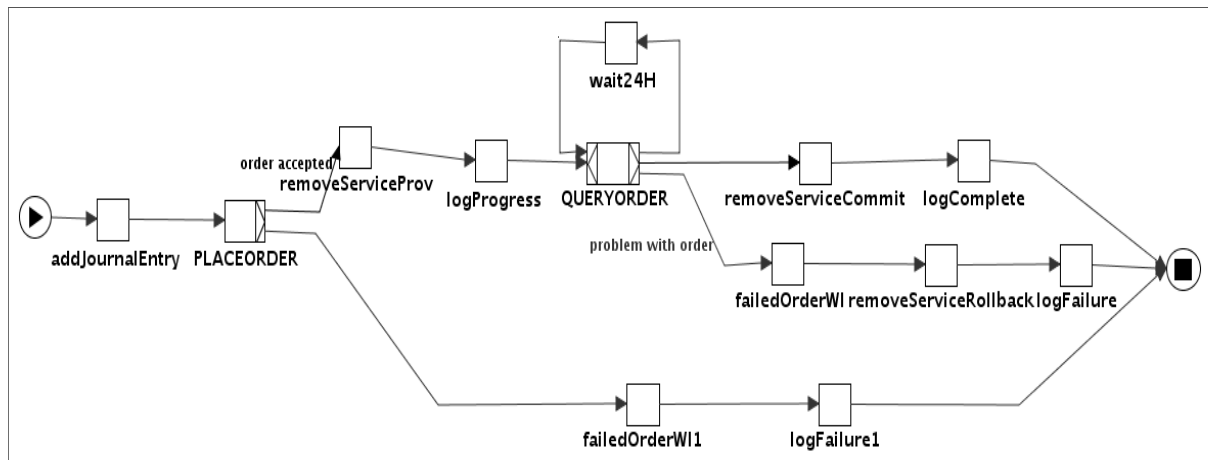

Figure 7.1: Process Model to Withdraw a WLR Service

followed by a successful result of the *PLACEORDER* task is indicated by the *order accepted* label on the relevant outgoing branch of the task.

Thus if the *PLACEORDER* task was successful, control passes to the *QUERYORDER* task which monitors the SPG system for completion of the order. As the lead-time for an order to complete is typically 10 working days, the looping branch with the *wait24H* task is used to query the outcome of the request once per day until confirmation is received. Once it is received, flow passes to the *removeServiceCommit* task and finally the *logComplete* task after which the process instance completes.

If a non-successful response is returned from the SPG to the *PLACEORDER* task, the process is routed to the *failedOrderWI1* task which appears in the task list of a user, who implements a "human-based exception handler" to cater for this instance of an expected exception. After the exception is user-processed and logged, the process instance then completes, so that a new instance of the process must be started to make another attempt to withdraw the telephone service for the customer as required. While the human exception handler is able to deal with a variety of exception types, the way they are handled are not explicitly recorded on the system, and so are not available to the system in the event of the same or similar scenarios happening during the execution of subsequent process instances.

It may sometimes be the case that, while a user's request for withdrawal of their telephone service is processing, they submit a request to cancel the withdrawal (for example, a customer may have sold their home and were to have moved out by a certain date, but the date has been extended or the sale has fallen through). In such cases, if a withdrawal order has been received but has not yet been completed, a new order (known as an "order cancellation order") must be

submitted that references the original order to be cancelled. This scenario is an example of a case-level external exception.

first:utility approached this situation by creating a secondary compensation process, which is initiated when the original withdrawal order process is cancelled by a call centre user following a customer request. The tasks performed by the compensation process in this case depend on the current status of the original withdrawal process, which may have one of five distinct states:

1. the process has begun, but the withdrawal order has not yet been placed on the SPG (that is, prior to completion of the *PLACEORDER* task).

2. post (1), but the database has not yet been updated to record the placement of the order (that is, prior to completion of the *removeServiceProv* task).

3. post (2), waiting for completion notification from the SPG (*QUERYORDER* has not yet completed).

4. post (3), but the database has not yet been updated to record the completion of the order (*removeServiceCommit* has not yet completed).

5. the process has completed.

Thus knowledge of the current state of the original process is required to effectively compensate for the cancellation of an order. This was achieved by interrogating the values of various net-level (i.e. case-level) variables of the original process to facilitate conditional branching in the compensation process, which produced the process schema *FixedLineWLRWithdrawCompensation* seen in Figure 7.2.

Five distinct paths exist in this compensation process. Conditional branching emanates from the *addJournalEntry* task; the appropriate branch is taken depending on whether the net-level variable containing the SPG order reference has been populated, and if so, whether the order has been completed on the SPG. If there is no order reference (state 1), the withdrawal order has not yet been submitted to the SPG, so no further action through the SPG is required and the process branches to the *removeServiceRollback* task. If the order has been completed (state 4 or 5), then it can't be cancelled and the process branches to the *failedOrderWI1* task (a user-worklist task notifying the cancellation request could not be actioned on the SPG).

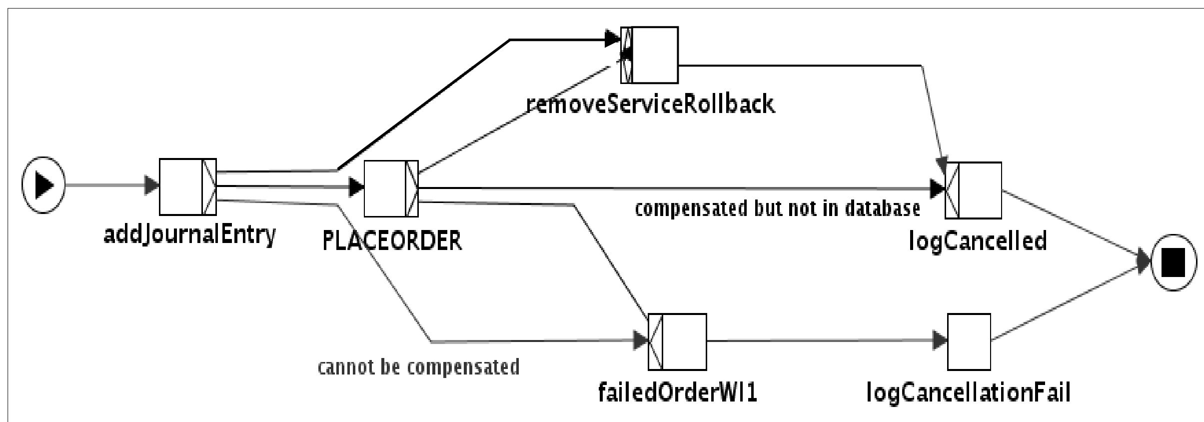

Figure 7.2: Process Model to Compensate for the Cancellation of a WLR Withdrawal Instance

If the order has been placed but is not yet completed (state 2 or 3) then the process branches to the *PLACEORDER* task, where the cancellation order is submitted through the SPG. Once submitted, the SPG may reply that the order has been received too late (within its own processes) to cancel the order, in which case the process branches to the *failedOrderWI1* task. If it is not too late to cancel the order (indicated by a successful response from the SPG), then processing continues to either *removeServiceRollback* (if state 3) or *logCancelled* (if state 2) after which the process completes.

Complicating factors in this approach were how to provide the compensation process with access to the net-level variables of the original process, and how to suspend an executing process pending the outcome of a compensation — neither capability being supported by the 'vanilla' YAWL environment. first:utility identified four mechanisms that needed to be implemented as extensions to the YAWL environment to support their compensation processes:

- the ability to suspend an executing process instance;
- the ability to resume a suspended process instance;
- the ability to obtain the current set of net-level variables of an executing process; and
- the ability to map the values of those variables to the net-level variables of a compensation process and launch that process.

The ability to suspend and resume a process instance was achieved by additions to the YAWL environment and made accessible to custom services through Interface A. Further, the configuration of process specifications was enhanced to enable a compensation process to be associated with another (standard) process.

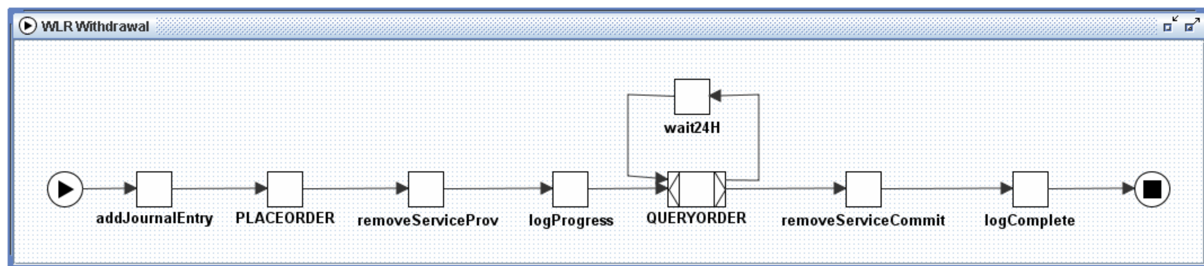

Figure 7.3: A Worklet Solution for the WLR Withdrawal Process

A requirement of the approach was that *all* of the net-level variable declarations and associated data type definitions present within the main process must also exist within the compensation process specification, such that the entire data set of the original process is transferred to the compensation process prior to the compensation process being launched.

In summary, first:utility have developed an effective method to deal with expected exceptions during the execution of their business process instances. By introducing a small number of extensions to the YAWL environment, processes may be manually suspended and cancelled, and compensation processes may be manually raised using the data values of the original process instance. Such processes require that all compensation actions are defined within the static process schema, with conditional branching explicitly modelled to facilitate the performance of the actions appropriate in each process instance.

7.1.3 A Worklet Service Solution

Using a worklet approach, the main process can be simplified by removing the exception handling branches so that only the primary business logic remains. Figure 7.3 shows how the schema for the main process can be defined using a worklet framework.

The tasks in the model now exactly correspond to the 'normal' control flow as listed in Section 7.1.2. By separating the exception handling tasks from the business tasks, the 'parent' process model becomes much cleaner and easier to comprehend.

The exception handling branches have been 'relocated' to become two compensatory worklets, each being a member of an exlet. The first exlet will be invoked following the failure of a ItemPostConstraint rule for the *PLACEORDER* task. That is, rather than testing an item-level variable's value in a conditional expression explicitly embedded in the process schema to determine which XOR branch to take after the *PLACEORDER* task completes, the post-constraint

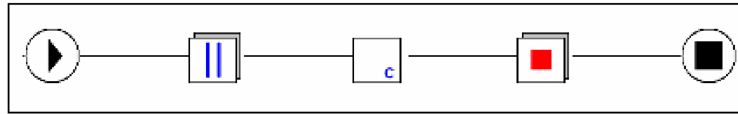

Figure 7.4: ItemPostConstraint Exlet for PlaceOrder task

rule check allows the test to be performed outside of the schema, thus totally removing explicit conditional branching from the main process. When the constraint fails the exlet, which consists of three primitives (Figure 7.4), is executed:

1. suspend case
2. run worklet OrderFailed
3. cancel case

In this way, the actions of suspending the case, executing the compensatory worklet and then cancelling the case have been fully automated and so no longer depend on manual intervention and action. In addition, the net-level variables of the process are automatically and constantly accessible to the worklet service (via the CaseMonitor object); only those variable values that are required for the operation of the compensation worklet are mapped to it — there is no requirement to map the entire data set of the parent process to the worklet.

The *OrderFailed* worklet is shown in Figure 7.5. When invoked, the worklet performs the necessary tasks to compensate the order failure, thus capturing the essence of the former approach. Of course, other exlets could be added to the repertoire for this task/constraint combination at a later stage that, for example, would modify the order (if possible) so that it could be resubmitted within the same case; such an exlet might resemble: {*suspend case; compensate; restart item; continue case*}.

The exception branch after the *QUERYORDER* task in the original *FixedLineWLRWithdraw* schema has some similarities with the one after the *PLACEORDER* task in that schema, except that the former involves an extra task to rollback the database entry for the withdrawal. Because

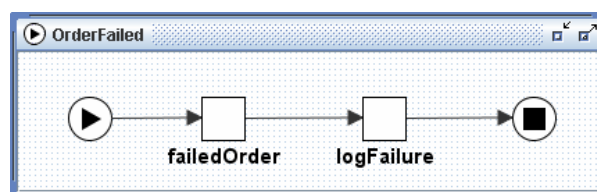

Figure 7.5: Compensation Worklet for a failed WLR Withdrawal Order

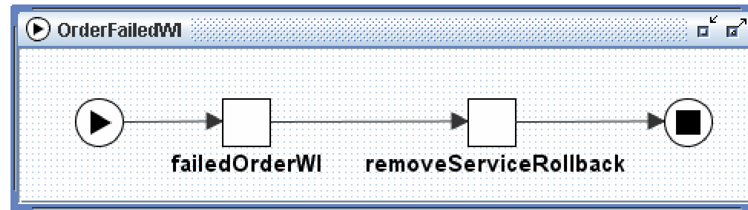

Figure 7.6: WithdrawOrderFailedAfterPlace Worklet

of those similarities, an ItemPostConstraint rule can be added to the *QUERYORDER* task in a similar fashion to the rule for the *PLACEORDER* task. In fact, the rule can be set to invoke the same compensatory worklet in both cases; the difference between them can be achieved by making the first task of the *OrderFailed* worklet, *failedOrder*, worklet-enabled. Then, by setting the selection rule of that task to substitute it with the worklet *WithdrawOrderFailedAfterPlace* in Figure 7.6, the single *failedOrder* task can be replaced with the worklet when appropriate (i.e. when it is invoked via the ItemPostConstraint rule tree of the QUERYORDER task).

Of course, the worklet framework also allows for the creation of two distinct compensatory worklets for a failed order, rather than a single one that can be modified via a worklet-enabled task and substitution. It can be seen here that the worklet approach delivers a great deal of flexibility to the designer/analyst that enables the ability to construct a set of processes and compensatory worklets that best fit the circumstances of the particular process (including business rules and the comprehensibility of the 'schema-set' amongst the organisation's stakeholders).

Another example of the flexibility in modelling choices made available in the worklet approach is the looping branch around the *QUERYORDER* task, which remains unaltered from the original schema to the worklet example in Figure 7.3. A choice was made to include the construct in the worklet example on the basis that it adds meaning to the graphical representation of the schema. Alternately, it could have been modelled as a separate one-task worklet (containing the *wait24H* task) and an additional ItemPostConstraint rule for the *QUERYORDER* task that, whenever the order was not complete, would run an exlet: {run waitWorklet; restart item}.

The static Cancel Withdraw Order compensation process *FixedLineWLRWithdrawCompensation* in Figure 7.2 contains five distinct control flow paths, corresponding to the compensation tasks that must be performed depending on the current state of the withdraw order (parent) process instance that it is compensating. They are:

1. {addJournalEntry, logCancelled} if the parent instance has begun but the withdraw order

has not yet been placed.

2. {addJournalEntry, PlaceOrder, logCancelled} if the withdraw order has been placed but the customer's database record has not yet been updated.

3. {addJournalEntry, PlaceOrder, removeServiceRollback, logCancelled} if the withdraw order has been placed and the customer's database record has been updated.

4. {failedOrderWI1, logCancellationFail} if the withdraw order has been placed and the SPG notifies that it is too late to cancel that order.

5. {failedOrderWI1, logCancellationFail} if the withdraw order has been completed.

Even for a small compensation process such as this, it is easy to see that, if other possibilities need to be accounted for and thus extra control branches added to the schema, it can soon become quite difficult to follow what the process is actually trying to achieve, which branches correspond to which process states, and so on. Also, whenever this compensation process is run, the parent process is first suspended and afterwards cancelled for every instance of the compensation process. Thus there is no option to manage the state of the parent process in any other way (besides suspend, unsuspend or cancellation) which may be more applicable to the actual circumstances of the particular process instance.

The worklet service, by providing a dynamically extensible repertoire, provides many advantages over a statically defined compensation process. First, the compensation process can be redefined as a set of modular components that can be invoked in certain combinations to manage the compensation for any particular state of the parent process. Second, new compensations can be added to the repertoire at any time, rather than being required to modify and extend the static compensation process. Third, an exlet definition, in addition to executing compensation processes, allows for a number of actions to be automatically applied to the parent item, case or specification (including suspend, cancel, restart, complete, fail and so on), or allow the parent to continue while the compensation is run, as the case requires, rather than a mandatory manual suspend then cancellation of the entire case. Fourth, the compensation worklet defined for a particular need can be reused in other circumstances as required, reducing the time spent on modification and redevelopment, compared to that for static compensation processes. Fifth, each added worklet is verified once only (when it is created); in contrast, a static compensation process must be re-verified every time it is modified, and verification becomes increasingly

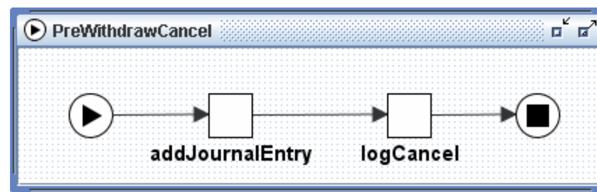

Figure 7.7: Compensatory Worklet PreWithdrawCancel

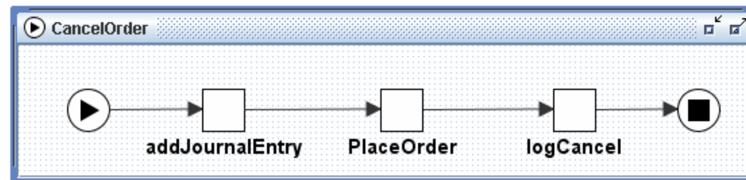

Figure 7.8: Compensatory Worklet CancelOrder

complex. Sixth, the framework provides for compensations for other external events that may affect the operation of the parent process (i.e. other than an order cancellation) to be added at any time.

A worklet solution in this example would deconstruct the original compensation process to become three simple worklets (Figures 7.7 – 7.9):

1. PreWithdrawCancel, corresponding to state (1) above;
2. CancelOrder, corresponding to state (2); and
3. CancelOrderWithRollback, corresponding to state (3).

Each of those worklets automatically become members of the repertoire available to the 'parent' withdraw order process. As such, there is now one process defining the business logic and control flow for a withdraw order case, and a number of worklets that will be invoked for selection and exception handling on an as-needed basis.

For the states (4) and (5) described earlier, both invoke the same two tasks in the original compensation process (Figure 7.2), so the same compensatory worklet can be used for each. Further, the *OrderFailed* worklet created for task substitution, shown in Figure 7.5, can be reused here to handle each state (the data values passed to the *failedOrder* task — the instance's context — would differentiate how the task is performed in each case).

It should be noted here that finer grain worklets could have been defined so that, for example, the centre task of the *CancelOrder* worklet, and the central two of *CancelOrderWithRoll-*

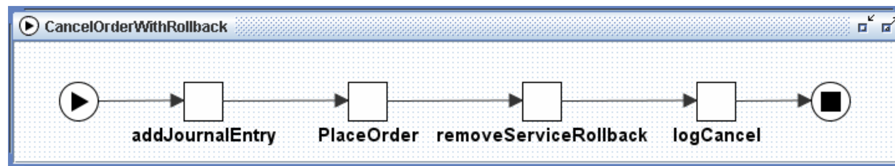

Figure 7.9: Compensatory Worklet CancelOrderWithRollback

back could be substituted with selection worklets by making the *addJournalEntry* task worklet-enabled. In this case, three non-worklet-enabled worklets were defined to again demonstrate the flexibility offered to the designer/analyst to decide to what level of granularity the schemas should be defined.

To correspond to the example provided in Section 7.1.2, all of the exlets defined for this process first suspend the case, then after the relevant compensatory worklet has completed, cancel the parent case. Of course, other exlets could be defined and added to the repertoire as required to, for example, modify the behaviour of the parent case so that it might continue after certain compensations have been run.

With regards to rule definitions, along with the RDR defined for Selection and ItemPost-Constraint within the parent Withdraw Order process in Figure 7.5, a rule tree is added for the CaseExternal rule type to invoke the relevant exlet depending on the state of the instance when an 'order cancellation order' request is received. In addition, the worklets *CancelOrder* and *CancelOrderWithRollback* each have a ItemPostConstraint rule added for their *PlaceOrder* tasks to invoke the *OrderFailed* compensatory worklet if the SPG notifies that the cancellation request has been received too late to be actioned.

Figure 7.10 shows the CaseExternal rule tree for the WithdrawOrder process. It can be seen that the situation where a specific action needs to be taken depending on the current state of the parent process lends itself very neatly to an RDR framework. Each subsequent node on the rule tree is an exception to its parent node (i.e. a more specific rule to its parent), and so the rule 'ripples down' to choose the most appropriate exlet depending on the current state of the parent process instance. The conclusion of each rule node shows a textual representation of the exlet to execute when that node is the last satisfied — graphically, each of the exlets in this case will resemble that in Figure 7.4, although the actual compensatory worklet differs in each.

Figure 7.11 shows the ItemPostConstraint rule tree, which is added to the rule set for each of the following tasks:

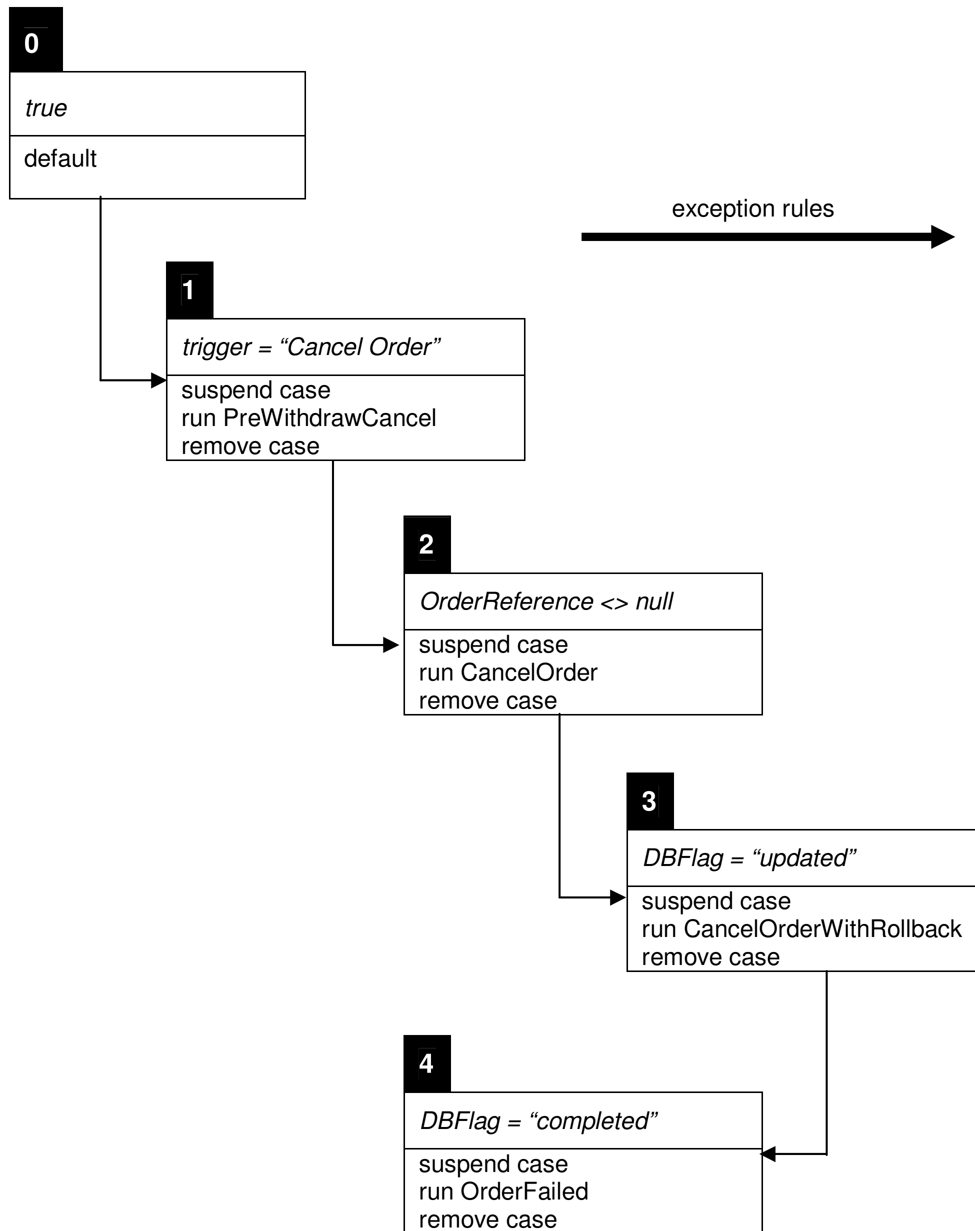

Figure 7.10: CaseExternal rule tree for the WithdrawOrder process

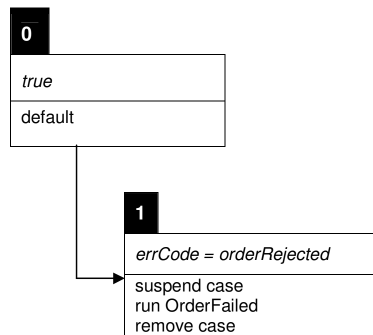

Figure 7.11: ItemPostConstraint rule tree used by multiple tasks

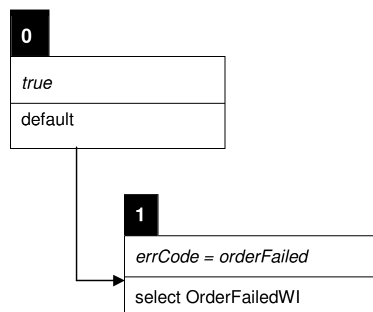

Figure 7.12: Selection rule tree for the failedOrder task of the OrderFailed process

- the *PlaceOrder* task of the *WithdrawOrder* process;
- the *QueryOrder* task of the *WithdrawOrder* process;
- the *PlaceOrder* task of the *CancelOrder* process (worklet); and
- the *PlaceOrder* task of the *CancelOrderWithRollback* process (worklet).

Finally, Figure 7.12 shows the Selection rule tree for the *failedOrder* task of the *OrderFailed* worklet (Figure 7.5). If the condition of rule 1 is satisfied, the task is replaced with the worklet *OrderFailedWI* (Figure 7.6); if it is not satisfied, *failedOrder* proceeds as an ordinary, static, atomic task.

This section has demonstrated many of the benefits offered by the worklet approach to facilitate flexibility and dynamic exception handling. A summary of the advantages of the worklet service discussed in this exemplary study are shown in Table 7.1.

The feedback from first:utility regarding the insights provided by the worklet approach in this exemplary study has been very positive. The worklet service was described as an “obviously flexible approach” with “robustness” coupled with a “high degree of flexibility”. They indicated that they could see clear advantages in the worklet approach which were of a kind not available

in any commercial products that they were aware of, and would install and try out the service in several other business processes in the near future.

7.2 Exemplary Study - Creative Environment

Film and television production is a multi-billion dollar industry. In Australia alone, there are over two thousand film and video production services businesses actively employing almost twenty thousand people [166]. However, the industry is extremely competitive and has become progressively global in its scope. Even though the work processes of the industry are highly creative and goal-oriented, organisations are increasingly recognising the value of more conventional business management strategies, such as workflow, to gain and maintain a competitive edge [113, 94].

That is not to say that any workflow solution is able to be applied across the board to support all aspects of a film production process. Some types of work practices are simply too unbound or ad-hoc to gain any real benefit from a workflow solution; artistic endeavour often means never doing the same thing twice. But there are many aspects of the industry where meaningful benefits can be gained through the use of a workflow solution to assist in the management of a project, including:

- back-office administrative and support processes;
- the allocation of resources to tasks;
- routing of film stock, documentation and other materials amongst employees; and
- facilitating inter-team communication and goal-setting.

This study will examine two processes that occur during the post-production phase and discuss the applicability of using a worklet-service-based solution to support those processes.

Rather than coming at the end of production (as the name might imply), work within the post-production phase operates concurrently with several other phases of production. Typically, footage produced each day is passed directly to post-production teams, so that post-production tasks are undertaken while subsequent filming continues. Tasks in this phase include the merging of video and audio components (voice, sound effects, music and so on) and editing to

Table 7.1: A Comparative Summary of Exception Handling Approaches

	first:utility	**worklet service**
Comprehensibility	All business logic and exception handling branches must coexist in the process model	Parent process shows business flow only; exception handling routines stored as worklets
Modularity of compensation processes	All compensation tasks, covering all anticipated situations, defined within a single process	Multiple, modular processes defined, which can be executed in flexible combinations
Dynamic change	Static process must be manually amended to accommodate change	New worklets, exlets and rules can be added to repertoire at any time
Modify state of parent	Suspend, unsuspend and cancel only	Full automatic control (suspend, unsuspend, cancel, restart, force-complete, fail)
Effect of state change	Parent case only	Able to modify state of parent case, workitem, all cases with same specification, all ancestor cases
Parallel execution of parent and compensation	No	Yes
Unexpected exceptions	Not directly supported	Supported - repertoires are extensible dynamically, even during execution of process instance triggering the unexpected exception
Execution of multiple compensations per exception	No	Yes
Multiple levels of granularity	One level only, with all tasks defined within it and multiple flow paths required	Multiple levels of granularity available to designer
Reuse	Each compensation process unique to its parent	Worklets easily reused in repertoire of other tasks and cases
Verification	Each change of static compensation process requires reverification of entire process	Each worklet verified once only
Rules and conditions	All rules explicitly modelled in schema using XOR splits	Discrete, dynamically extensible rule base external to enactment engine
Rules evaluated	By enactment engine as part of schema execution	By worklet service, external to enactment engine
Data mapping from parent to compensation process	All parent net-level data definitions, declarations and values must be exactly duplicated in compensation process	Only relevant data definitions need be defined in compensation process; data can be mapped from net-level of parent, workitem-level or from external source
Data mapping from compensation to parent process	Not supported	Updated values mapped back to parent when worklet completes
Data sources	Net-level values of parent	Net-level values of parent, workitem-level values, internal status of each workitem in case instance, resource data, historical data from process logs, and other extensible sources

produce a coherent, final piece, and as such includes tasks that are both acutely technical and highly creative. Often, there is a struggle to find a balance between creative license on the one hand and time and cost constraints on the other [111].

Because of the depth of the tasks to be completed in post-production, there is a high level of complexity involved in the techniques used and the order in which they occur — as such, the way tasks are completed may be said to be unique for each production. The post-production phase is also heavily affected by the rapid changes in technologies that occur; for example, the digital methods used today bear no resemblance to the methods employed as little as five years ago.

In addition, participants, who are primarily focussed on their own tasks, are often not aware of the complexities involved in the 'bigger picture', and are ambivalent towards the perceived benefits that a workflow solution may bring. However, by modelling the processes and providing a workflow solution that supports creative and ad-hoc processes, it can be demonstrated that efficiencies can be found and organisational costs reduced.

The process models provided to this research for the study take the form of Configurable Event-Driven Process Chains (C-EPCs), a language that extends from the Event Driven Process Chains (EPC) language to provide support for the configuration of Enterprise Systems [150]. C-EPC models capture potential configuration alternatives for systems and provide decision support for the selection of process variants. Thus, they show options available and decisions to be made at design time to individualise the generic model for a particular system.

The processes originate from a cooperative project between the *Queensland University of Technology* (QUT) and the *Australian Film, Television, and Radio School* (AFTRS) within the context of QUT's *Centre of Excellence for Creative Industries and Innovation*. The general descriptions of the C-EPC process models are based on similar processes as discussed in [121].

7.2.1 Process: Post Production

The first process examined is an encapsulation of the entire post production process; it is referred to as the *Master* process. Figure 7.13 shows the C-EPC representation of the process (adapted from [121]), which can be logically divided into three phases: pre-edit, edit and post-edit.

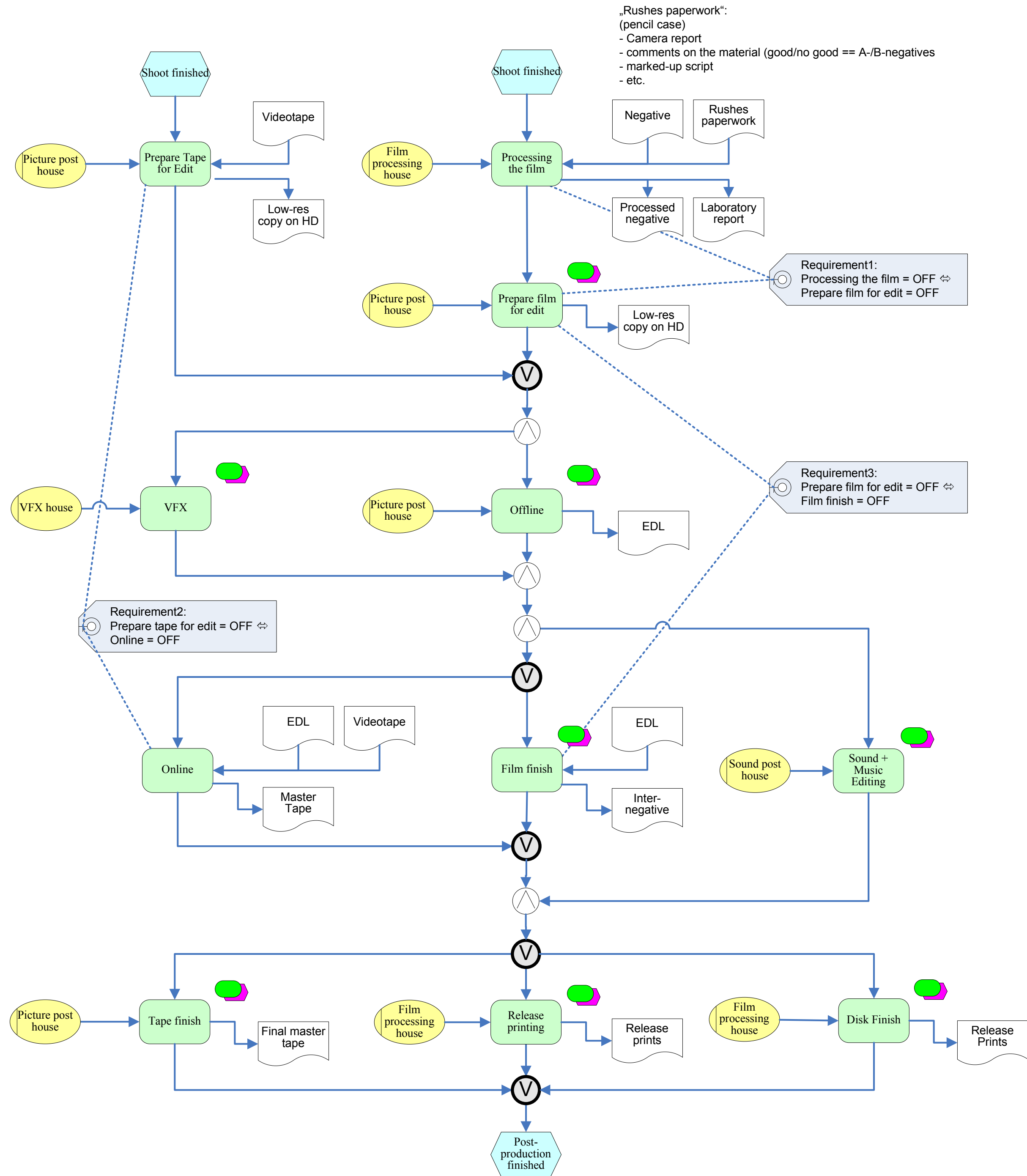

Figure 7.13: Static Post Production Master Process (C-EPC language) (adapted from [121])

Pre-Edit The first or pre-edit phase begins with the delivery of the day's footage ('the rushes') to the post-production team. There are two possible entry points into the process, one for each type of media that may be used (film and videotape). It may be the case that both types of media are used for a particular set of rushes, so in terms of the model's entry points, inputs may arrive at both simultaneously. Film media is delivered as an exposed negative which must first be processed, then delivered to the picture post house for editing. Videotape does not require the same processing step as film, so goes directly to the picture post house. For both media types, a low resolution copy is digitised and stored on computer file to be used as a guide for the remaining process.

Accompanying the film and/or tape is the 'rushes paperwork', a set of documentation which may include items such as an annotated script, and video and audio reports. This documentation is regarded as an important source of information about the footage, and is thus made available throughout the post production process.

Edit In the edit phase, the video and audio components are handled separately. Further, video editing is divided into low and high resolution edits. Low resolution editing is represented by the *Offline* task, which allows editing decisions to be made and documented before the high resolution editing begins. The result of the *Offline* task is the EDL (Edit Decision List). Video Effects Production takes place concurrently to the *Offline* task (see Section 7.2.2 for a detailed discussion of the VFX process).

When the *Offline* task completes, the high-resolution editing, along with the sound and music editing, can begin. For film, the high resolution editing takes place in the *Film Finishing* task, where the original negative is spliced into pieces, some of which are rejoined; for tape, it occurs in the *Online* task, where the video is rearranged using an editing suite and recorded to a tape master. Both take the EDL output from the low resolution edit and each performs the actions listed in the respective EDL on distribution quality media.

Post-Edit After the edit phase, an edited, high resolution, distribution quality film and/or tape, together with completed visual effects and sound and music, is completed, and now must be 'finished' for distribution. The finishing may be required for any or all of the film, tape and disk mediums, which are output in the form of a release print, master tape or release version respectively.

The C-EPC representation of this model reveals some of the complicating factors that come in to play when rendering this process to a particular modelling framework. For instance, the C-EPC model contains 'requirements' or constraints which are designed to remove some tasks from the eventual process if certain preceding tasks were not included in a particular configuration of the process[1]. For example, Requirements 1 and 3 will cause the removal of the *Prepare Film for Edit* and *Film Finish* tasks if the rushes were not received on film; similarly, Requirement 2 will remove the *Online* task if tape media was not received. In addition, because there are two entry points, there are three possible media combinations that may start a case instance (i.e. tape, film or both tape and film), and so the model requires a number of OR splits and joins to accommodate the various combinations and the tasks they entail.

Transferring the entire C-EPC model to the YAWL language shows similar complications in the describing the process and is possible flow paths via a static model (Figure 7.14). There are several OR splits and joins (to emulate the C-EPC representation the OR splits and joins are unnamed); conditionals are required to be embedded into each OR split output arc to determine whether they 'fire' or not. All are basically dependent simply on which media formats have been supplied to the process. For example, the first OR split task controls whether one or both arcs fire (one for tape, one for film); the OR split preceding the *Online* and *File Finish* tasks has a similar function, and so on. Thus in static representations, the control flow logic is embedded into the business process logic. As a result, it is not obvious from the model which path may be taken during a particular instance.

With the flexibility mechanisms available with the worklet service, the process can be modelled without much of the complexity, particularly by negating the need for the OR splits and joins. Figure 7.15 shows the worklet-enabled process. Immediately apparent is the fact that, in this case, all of the OR splits and joins have been removed from the process model, and therefore only of actual business logic remains.

The first task in the process, *PrepareForEdit*, is worklet-enabled. Associated with this task is the Selection rule tree shown in Figure 7.16. The rule tree shows that, if either tape or film has been supplied, the corresponding rule will be satisfied and the service will launch the appropriate worklet for that media. If the rushes have been delivered on both film and tape media, node 2 will be last satisfied, resulting in the launching of two discrete worklets, one for each medium (the two worklets are shown in Figure 7.17). Note that the conditional expressions

[1] After configuration, the result of a C-EPC model is an EPC model.

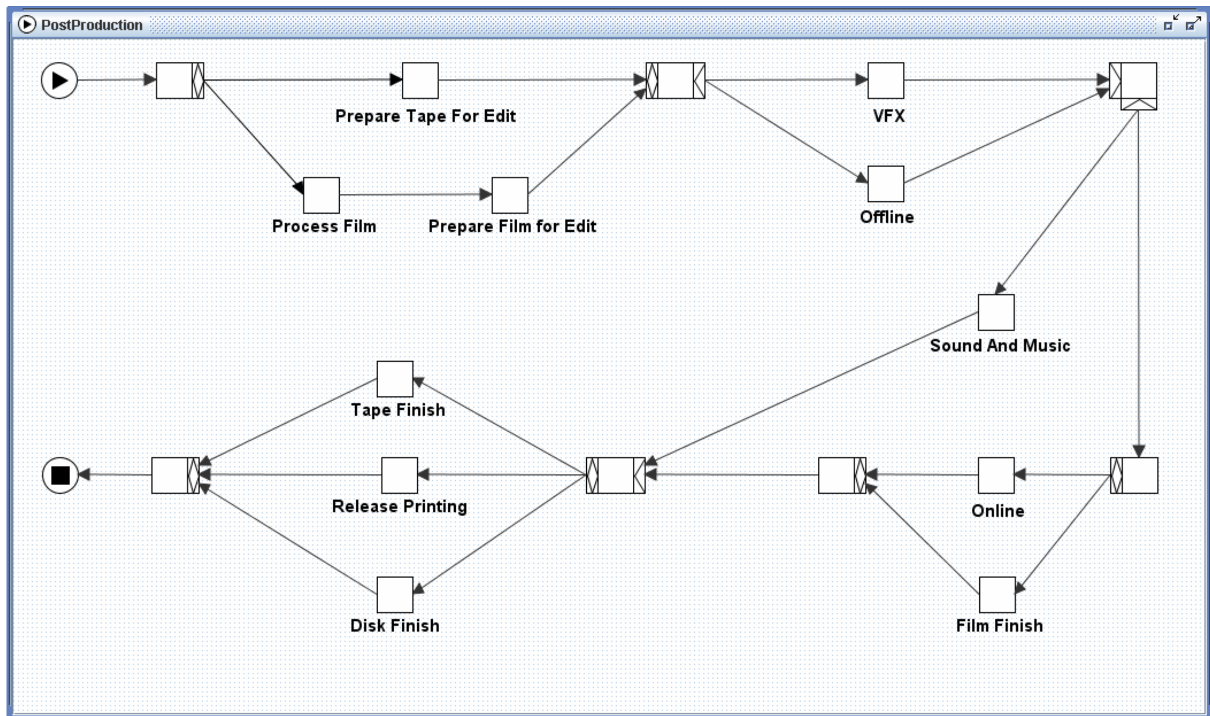

Figure 7.14: Static Post Production Master Process (YAWL language)

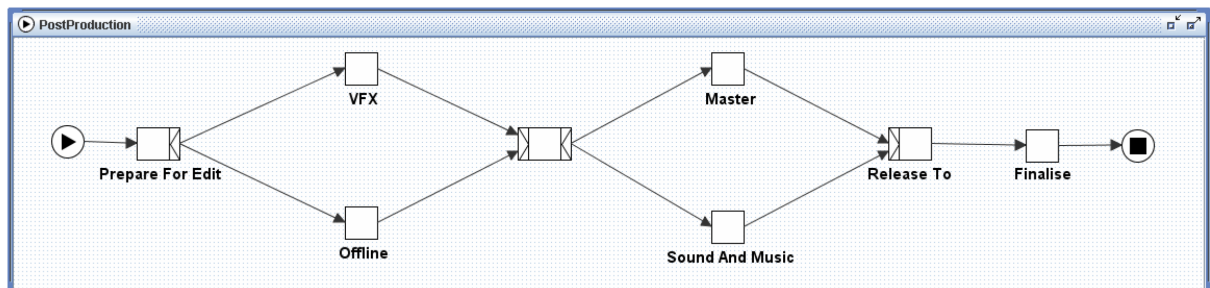

Figure 7.15: Post Production Master Process (Worklet-Enabled)

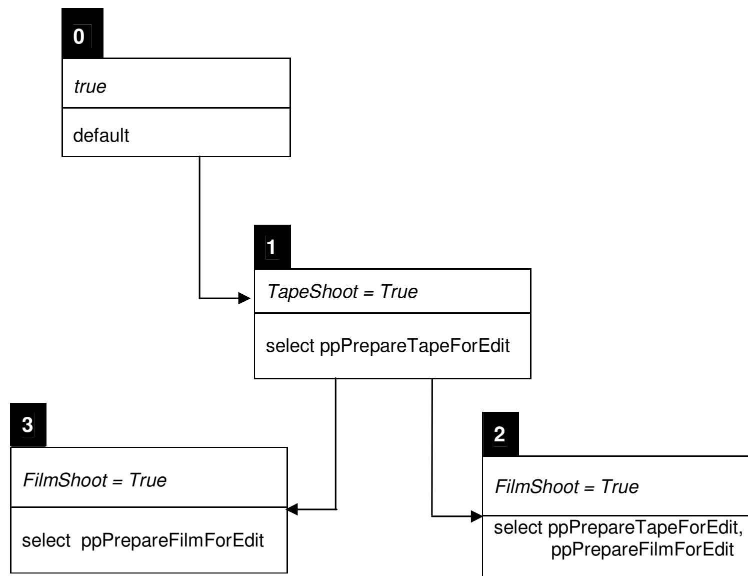

Figure 7.16: Selection Rule Tree for the PrepareForEdit task

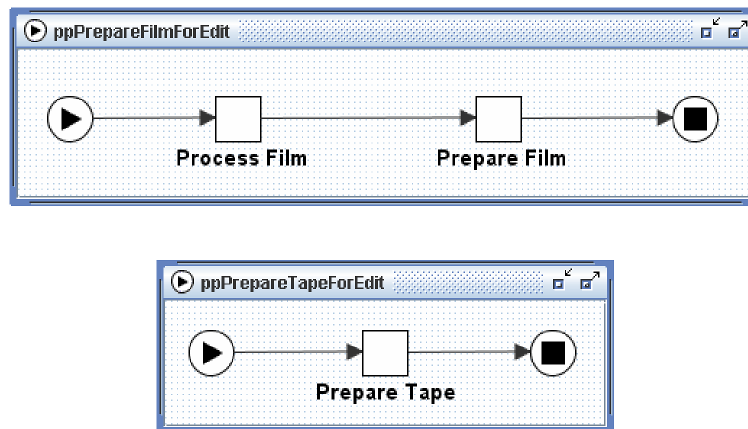

Figure 7.17: Worklets ppPrepareFileForEdit and ppPrepareTapeForEdit

for nodes 2 and 3 are identical in this tree, but their conclusions differ — node 3 will be tested if node 1 evaluates to false (i.e. there is no tape media), while node 2 will be tested if node 1 evaluates to true.

The worklet service allows any number of worklets to be concurrently launched as the result of a selection process. For exceptions, a compensation primitive can also launch any number of concurrent worklets; it also provides for any number of worklets to be run consecutively by inserting a sequence of compensation primitives into the exlet. Of course, combinations of concurrent and consecutive worklets may also be defined.

The worklet-enabled *Master* task performs a similar service to *PrepareForEdit* — it will launch a worklet to carry out the *Online* process if tape media is provided, and/or for the *Film*

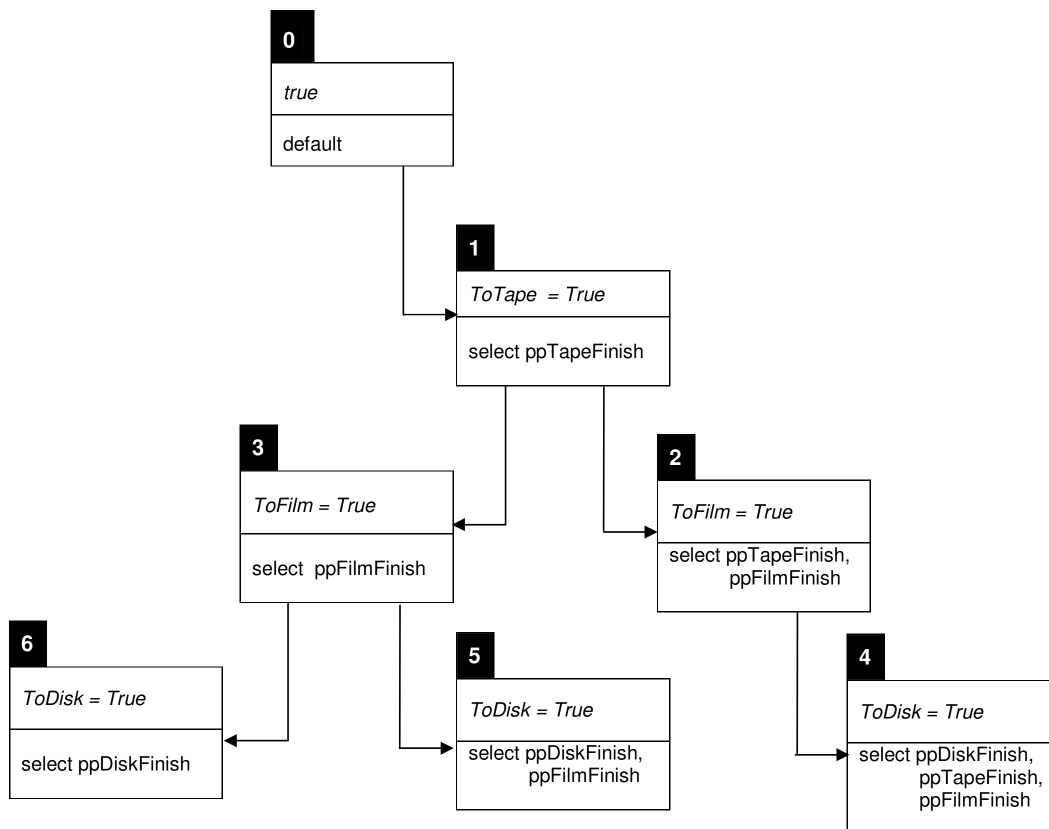

Figure 7.18: Selection Rule Tree for the Finalise Task

Finish process if film media is provided.

The *Finalise* task models the processing of 'finishing' the output for distribution. There are three possible sub-processes to perform: tape finish, disk finish and release printing — each has a corresponding worklet in the specification's repertoire. Therefore, there are six possible worklet launch combinations, as specified in the selection rule tree for the *Finalise* task (Figure 7.18). Each worklet consists of one task, corresponding to each of the three tasks at the post-edit end of the original static process model.

While there appears to be a pattern in the rule tree in terms of conditions being identical at each level, it is not necessary that this is the case. For example, nodes 3 and 6 could have been defined in reversed positions without changing the outcome of particular tree traversals[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 [156].

In summary, the worklet service allows a parent or master process to be defined without much of the explicit branching mechanisms necessary in the control flow of static models.

[2]A number of algorithms exist for the reordering and optimisation of rule trees; for example see [74, 146, 156]

As a result, the parent process models are cleaner, easier to verify and maintain, and easier for stakeholders to gain an understanding of the process logic. Once the parent process is worklet-enabled, it is able to access all the features of the worklet paradigm, including support for exception handling. Some exceptions that may occur in a post production process include damaged film or tape stock, equipment malfunctions and breakdowns, and time and budget overruns, to name but a few. All of these exceptions may be handled by adding an appropriate exlet to the specification's repertoire. Thus, it can be seen that, even for quite creative processes, the worklet service provides flexible and extensible support.

At the far end of the creativity spectrum, the functionality inherent in the worklet service is able to provide a virtual 'black-box' task, which can be constructed to receive data inputs from a variety of sources, and have myriad assortments of potential worklets available in its repertoire. Thus, the actual work process to be performed in a particular instance is late-bound to the process at the latest possible time after availing itself fully to the corporate memory and knowledge management domains of the organisation.

7.2.2 Process: Visual Effects Production

The parent post production process encapsulates a number of sub-processes, one of which is Visual Effects Production (VFX), which involves the planning, design and development of visual effects (VFX) for a filmed scene. Figure 7.19 shows the original visual effects production process model expressed in the C-EPC language. The process begins with the provision of a script and moves into a sequence of planning and design tasks which produces a design and storyboard of the planned animated scene. From there, the storyboards are passed to the visual effects team, who breakdown the scene into individual tasks and allocate them to team members.

The process then moves through four stages, which may be referred to as *Modelling*, *Painting and Texture*, *Rigging and Skinning* and *Animation*. Following those stages, any necessary live action sequences are shot, concurrently with any required motion capturing (mocap) — where live actors wear markers on various parts of their bodies, then move in designated ways so that the movements are digitally recorded and used to model the movements of animated figures — before the product of each is composited. Then, any required rotoscoping — where animators trace, paint or texture live action frame-by-frame — is performed before final digital

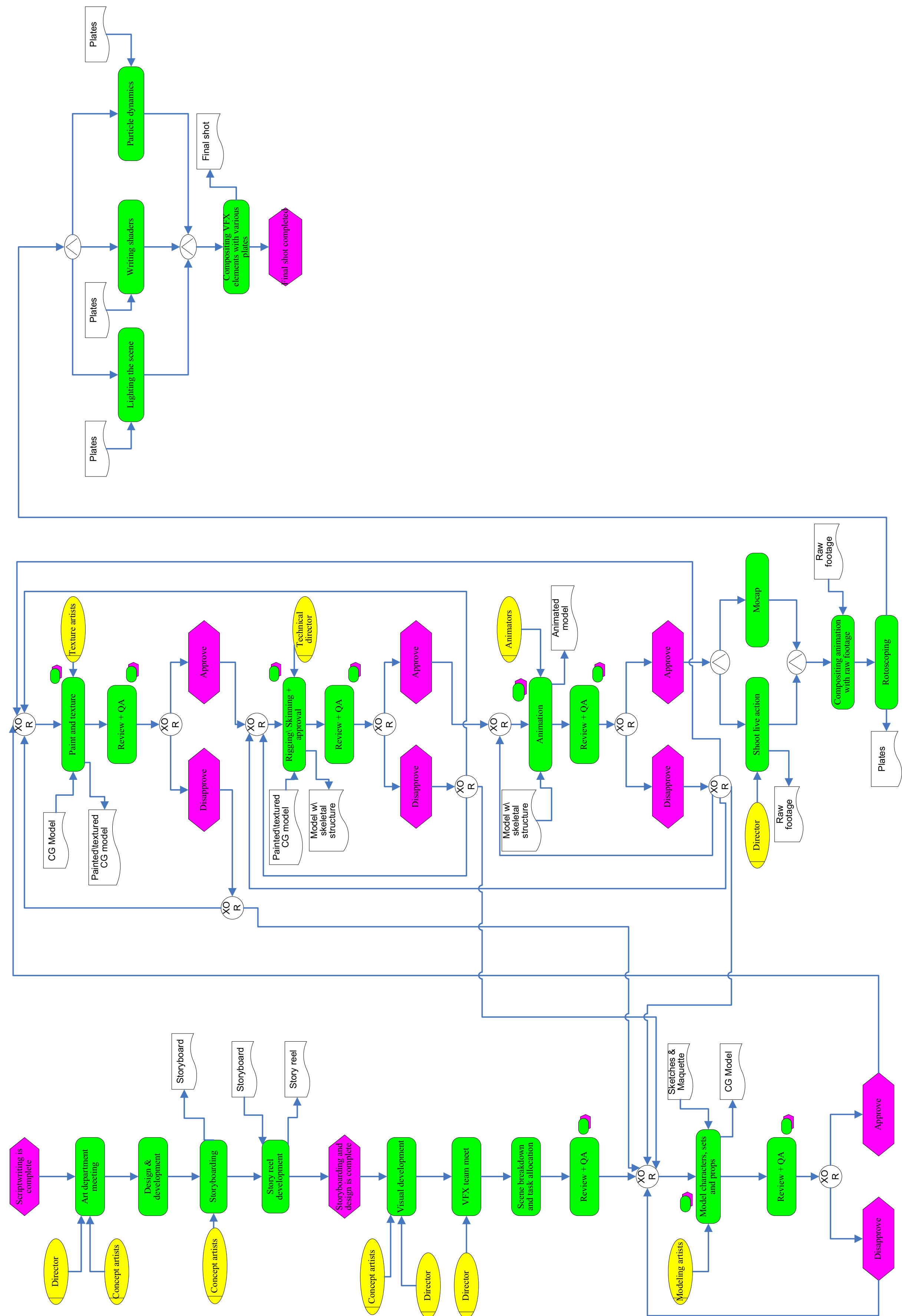

Figure 7.19: Static Visual Effects Production Process (C-EPC language)

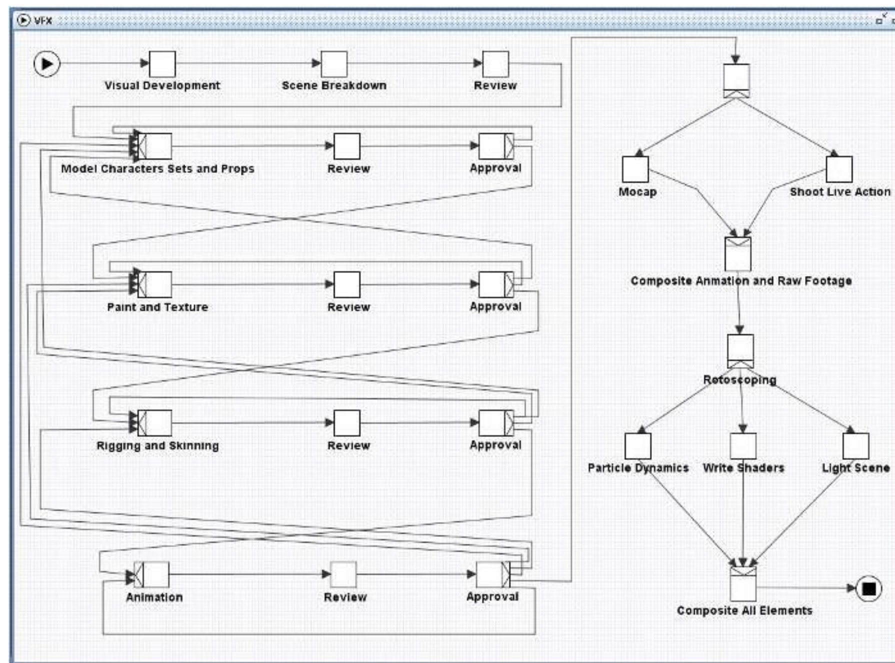

Figure 7.20: Static Visual Effects Production Process (YAWL language)

effects are added (lighting, shading, particle dynamics) and the scene is completed.

Even though this process schema models a very creative process, it is interesting that many parts of it are performed sequentially, making those parts ideal candidates for workflow support. However, the modelling of the control flow of the four central stages mentioned above adds a great deal of complexity to a static model. While each of these four stages must be completed in strict sequence, a decision is made at the end of each whether to continue to the next stage (if the work done at that stage is considered complete and satisfactory), or to return to a previous one (if it is deemed incomplete or unsatisfactory). That is, at the completion of each stage, process control flow may pass to the next stage, or to the start of the stage just completed (i.e. a redo of the current stage) or return to the stage of *any of the stages preceding it*.

The complexity this adds to the model can be seen in Figure 7.19 by the various branching that occurs from an XOR split following the 'disapprove' decision at the end of each stage to one or more XOR joins, one preceding each stage. For example, the XOR split immediately preceding the 'Model Characters, Sets and Props' task has five incoming arcs; the XOR split following the fourth stage 'Animation' has four outgoing arcs. This combination of XOR split and joins with the accompanying arcs adds some difficulty in tracing control flow for a particular process instance.

Figure 7.20 shows the YAWL static model equivalent of the same process; the first few

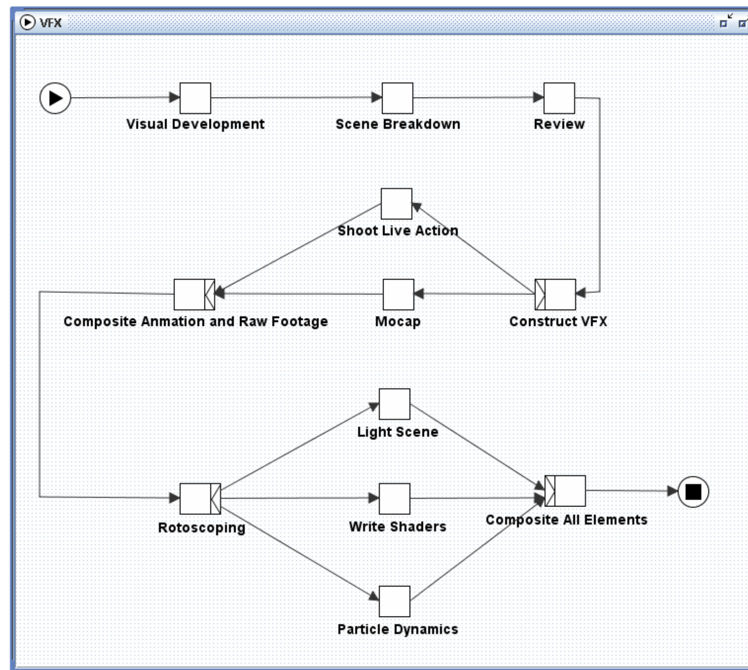

Figure 7.21: Visual Effects Production Parent Process (Worklet Solution)

sequential tasks have been omitted since their modelling is unremarkable. Here, the decision made at the end of each of the four stages is modelled by outgoing arcs from an XOR split decorating an *Approval* task in each. Again, the model is quite complicated in its definition, difficult to follow and verify for correctness, and any amendments made to it (for example, to add extra tasks or stages, or to modify the control flow between stages) would be particularly intricate and involved.

However, by using the functionality made available by the worklet service, much of the complexity is removed. Figure 7.21 shows a worklet solution for the same VFX process. Here the conditions and associated branches have been defined, not in the control flow of the parent process, but in an associated rule set, which automatically makes decisions for each case instance, based on its context, and executes the relevant worklet accordingly. The four stages discussed above have been replaced in this model by the single worklet-enabled task *ConstructVFX*, and each of the stages has been modelled as a discrete worklet, as shown in Figure 7.22. The rule set for the *ConstructVFX* task manages the ordered execution of the worklets, which ensures they are completed in the correct sequence, and that a disapproval at the end of a worklet will reinvoke the worklet of a previous stage as required. When all four worklets have been approved, the parent process continues.

The *Approve* task in each worklet works in conjunction with the rule set to determine

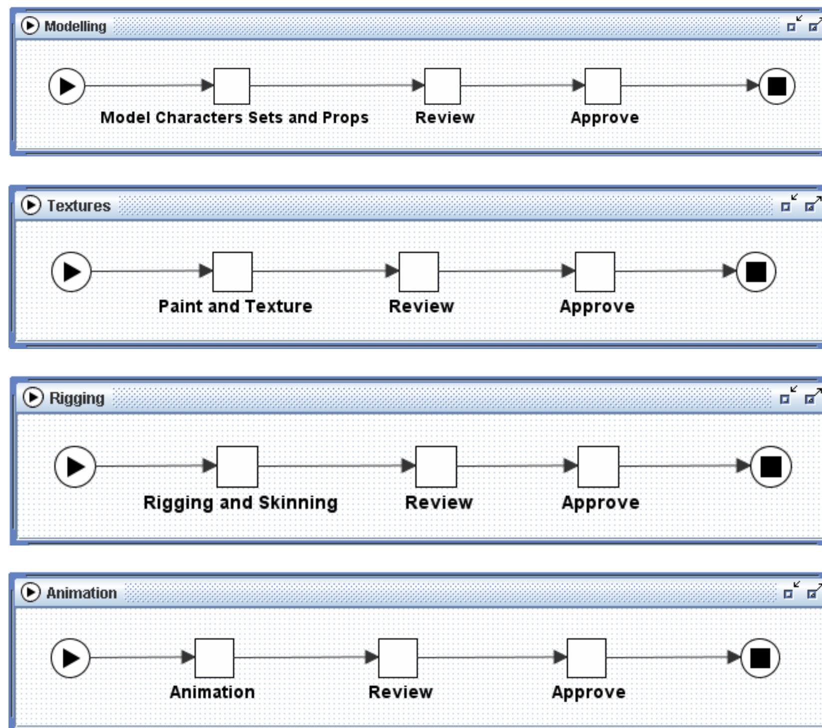

Figure 7.22: Visual Effects Worklets

whether to proceed or re-invoke the current or a previous worklet, and if so, which previous worklet to re-invoke. The task consists of a simple form where a choice from a number of options can be made. An example of such a form can be seen in Figure 7.23, which is generated for the *Approve* task of the *Animation* worklet, and has five possible choices (effectively: continue, redo this worklet, or redo one of the previously completed worklets).

When the *ConstructVFX* task is enabled, the worklet service will substitute it with the *Modelling* worklet in Figure 7.22. At this stage, the worklet is invoked through the selection service, and so acts as a substitution for the task and not as a compensation. To facilitate progress through the four worklets, each has a CasePostConstraint rule tree defined. Each rule tree will examine the choice made in the *Approve* task for that worklet and execute the worklet corresponding to the choice. Thus, when the *Modelling* worklet completes, its CasePostConstraint rule tree is queried and, if the *Modelling* stage is approved, the service will invoke the *Textures* worklet; if not, it will invoke a new instance of the *Modelling* worklet, passing to it as inputs the output values of the completing worklet. This process will continue, with each worklet being invoked as many times as necessary, until such time as the *Approve* task of the *Animation* worklet captures a choice to 'Approve and Continue'. At that stage, since there are no further worklets to be invoked, the worklet service will have completed its handling of the original *Con-*

Figure 7.23: Approve workitem of Animation Worklet in YAWL Worklist (detail)

structVFX task, and so will check it back into the YAWL engine, allowing the parent process instance to continue.

Figure 7.24 shows the complete set of rules for each worklet (as seen in the Rule Editor's 'Effective Composite Rule' panel). It can be seen that the only rule that needs to be defined for the parent *VFX* process is one selection rule for the *ConstructVFX* task — further, its conditional expression is simply 'True', since the *Modelling* worklet is to be invoked with every instance of the parent process. The CasePostConstraint rule tree of each worklet is used to invoke the required worklet as indicated by the choice made in the relevant *Approve* task. Each, except *Animation*, has a default 'else if True then...' rule to pass control flow to the next worklet in the nominal sequence (i.e. if there has not been a choice made to redo a previous stage). The *Animation* worklet needs no such rule, since if no redo choices have been made, no further worklets are invoked and so the process is allowed to continue to the task following *ConstructVFX*.

The worklets invoked by the CasePostConstraint rule trees are considered by the worklet service to be exception handling compensation worklets, even though in this case they are being used to cycle through a set of stages an unknown (at design time) number of times. The commencing selection rule on the *ConstructVFX* task checked that task out of the engine, so that the parent process will not be progressed by the engine until such time as the task is checked back into the engine. Effectively, the parent process is in a wait state until such time as *ConstructVFX* completes. The worklet service will not consider the task to be completed, and thus ready to be checked back in, until such time that all the compensations running for it have completed. As a consequence, since there is no requirement to alter the state of the parent process (such as suspending it) each of the exlets in the CasePostConstraint rule trees contain exactly

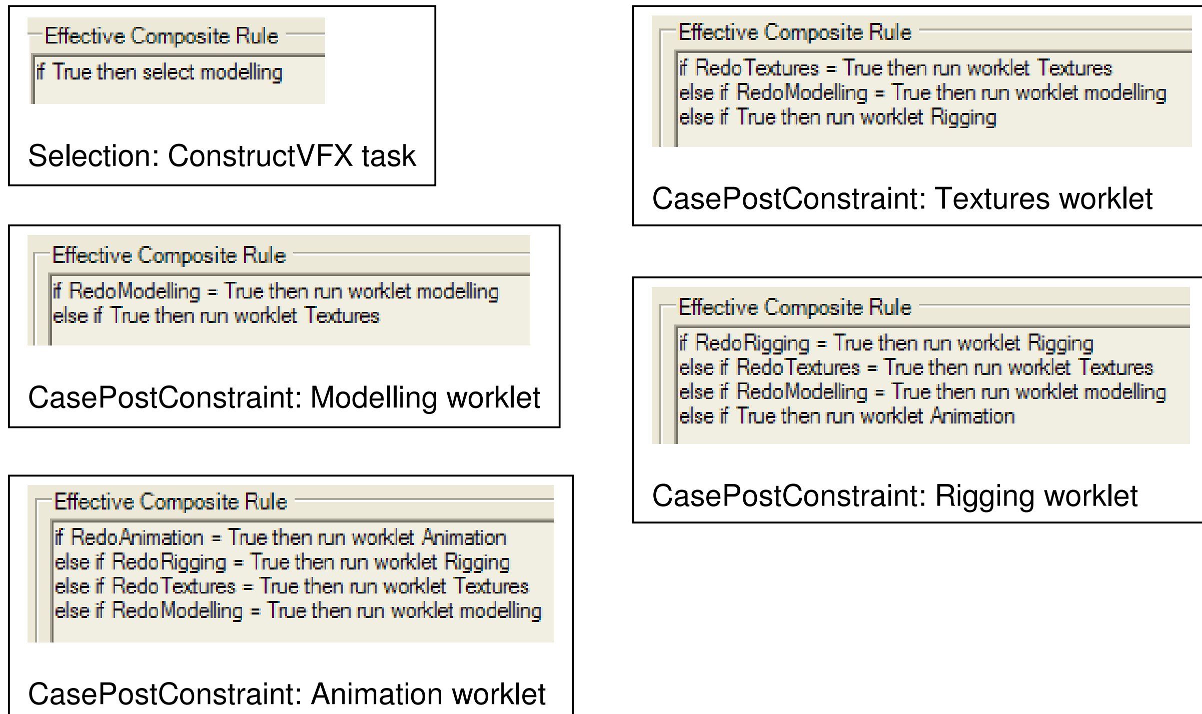

Figure 7.24: Rule Sets for VFX Process and Repertoire Worklets

one compensation primitive that invokes a single worklet.

As previously mentioned, once the rule sets have been constructed for the process and associated worklets, they can be easily added to, new worklets developed for tasks, individual worklets modified to capture new tasks as technology changes, exceptions catered for and so on. Therefore, while this exemplary study is an illustration of how the mechanisms offered by both the selection and exception service can be used in tandem to support flexibility in addition to pure exception handling, and so enables the definition of process models to be greatly simplified and more easily understood and maintained, it does not preclude this process for benefitting from the other advantages the worklet service has to offer.

Conclusion

The exemplary studies carried out in this chapter have demonstrated that the worklet service offers a number of advantages for workflow support in both business and creative environments. The processes used model real-world work practices and so validate the use of the worklet service as a viable support system. Improvements in flexibility, exception handling and process expression and extensibility have been established.

Chapter 8

Conclusion

This thesis began by identifying four key problem areas that describe the fundamental limitations of current workflow technologies with respect to the rigidity enforced by the inflexible frameworks employed, and the consequent difficulties in placing more dynamic, information intensive processes within those frameworks. Nine success criteria were then nominated that, if satisfactorily addressed by a workflow framework, would provide an effective solution to the problem areas identified.

Then, a study of Activity Theory was undertaken with an objective to providing a theoretical underpinning that reflected how human activities were actually performed. From that study, ten principles were derived that represent an interpretation of the central themes of Activity Theory applicable to understanding organisational work practices. Those principles were then mapped to six functionality criteria that this research proposed a workflow management system would need to meet in order to support the derived principles, and thus be considered to be based on a framework that supported the ways people actually work rather than on an inflexible proprietary framework.

It was found that the functionality criteria based on the principles of Activity Theory mapped rather neatly to the success criteria proposed in Section 1.3, which therefore served as a validation of the conceptual approach described. Figure 3.4 related the derived functionality criteria to the success criteria.

Based on the derived principles of Activity Theory, the worklet service was then conceptualised, formalised, implemented and validated. To ensure that the service meets the requirements of the derived principles, and therefore better supports the ways work is actually per-

formed, the service's outcomes are applied *below* to the functionality criteria developed from those principles. By meeting those functionality criteria, the success criteria described in Chapter 1 will also be met (through the mapping of criteria shown in Figure 3.4) and thus the key problem areas identified will have been successfully addressed.

Flexibility and Re-use The worklet service supports flexibility in several ways. Firstly, by providing a repertoire of actions (i.e. worklets) for each workitem in a process instance, a contextual choice can be made at runtime on which particular worklet to execute as a substitute for a workitem that best meets the needs of the individual instance. Further, if a particular choice of worklet is deemed inappropriate for the current case, then a new worklet/rule combination can be created and added to the repertoire *during case execution* and the case instance can take advantage of that newly available worklet.

Secondly, it has been demonstrated that the constraint checking rules incorporated into the exception sub-service can be used to provide natural deviations to the nominal control flow of process instances. These constructs also allow for an extensible repertoire of worklets to be made available to processes that add increasing flexibility to otherwise static process models, while avoiding the usual problems of versioning and migration control.

By using a modular approach, the definition of a specification may range from a simple skeleton to which actions can be added at runtime, thus supporting dynamic adaptation for those environments that have very loosely defined work practices or simply do not know at design time what path a process may take, or may be a fully developed model representing the complete current understanding of a work process, depending on user and organisational needs.

The worklet paradigm also provides for reuse, since one worklet may be a member of several repertoires and so only needs to be defined once. Indeed, a worklet may even launch another instance of itself if required. The modular approach offered by the worklet service enables repertoires of any size for a workitem, case or specification, and members of those repertoires to be as generic or specific as required.

Adaptation via Reflection The worklet service frees the designer/analyst from the rigid frameworks made available by proprietary systems. It allows plans (i.e. process specifications) to be fluid and extensible, and thus be executed in anticipation of a goal rather than represent a prescriptive map that must be followed in every case instance. Providing this extensibility allows

each process instance to be regarded as a potential learning experience, where a side result of each execution is further knowledge of how the work is performed in certain (possibly unanticipated) contexts.

By providing the means to record deviations, through the addition of new rules and corresponding worklets/exlets, the worklet service allows designers and users to reflect on how their work is performed and add those reflections to future instantiations of the process. By incorporating those reflections implicitly into the process specification, all deviations are handled and recorded on-system, so that in future instantiations the worklet service itself reflects on the recorded 'organisation memory' that has developed in its rule sets over time.

Thus, an assisted learning system is provided, where each addition to the rule sets results in a more refined 'understanding' of the work practices being supported.

Dynamic Evolution of Tasks Through the support offered by the worklet service, workers are no longer 'straight-jacketed' into tasks that never change. That is, context-tunnelling, where workers tasks becomes so regimented that they lose sight of the overall objective of the activity and thus suffer a decrease in motivation, is avoided in the worklet paradigm by offering workers the ability to see efficiencies and variations in the way they perform their tasks and have those variations implicitly incorporated into the process specification via an addition to a task's repertoire. Therefore, over time, a particular task dynamically evolves as its repertoire grows.

Because a worker is provided a mechanism which varies the way tasks are carried out, based on their context, it will be less likely that they 'disconnect' from the overall objective of the activity, but will retain some ownership of both the task they are invested with and an appreciation of their responsibility within the activity. Therefore, because the worklet service is built on a framework that reflects the way work is actually performed, workers will engage in their activities, rather than being resistent to regimentalised work patterns.

Additionally, tasks are able to naturally evolve with changes in technology, business rules, customer service and so on, without any requirement to re-work the original parent workflow process model.

Locality of Change The principles derived from Activity Theory nominate that the centre of change lies with the individual charged with performing a particular task. Therefore, change emanates at the local level and so supporting systems must allow for local change to be easily

incorporated.

The worklet service provides for change to be initiated at the level of the individual worker, by allowing them to reject a worklet as inappropriate within the context of a particular case instance. Since part of the rejection process requires a worker to nominate a possible method of dealing with the context of the case, they are able to take ownership of their own work and thus exert a level of control over their own work practices, while remaining aware of their role in achieving the overall goal of the activity.

From a system perspective, change is implicitly incorporated into a process specification each time a worklet/exlet is added to its set of repertoires. Thus change is indeed managed at the local level, such that the original parent process does not need to be modified. And because each worklet is a complete workflow process specification, verification can be easily performed at the local level and will not impact on other worklets in the repertoire. Also, because worklets can be members of several repertoires, verification only needs to be done once. This is not achievable with monolithic modelling paradigms.

Comprehensibility of Process Models Specification complexity is reportedly a major limiting factor in the uptake of workflow support systems [137]. This thesis has shown that the worklet service provides the means to remove much of the complexity from process models. By providing repertoires of worklets and exlets for tasks and cases, the parent process specification can be freed of the conditional control flow logic that often clouds the business logic which the specification has been designed to support. Thus, parent process models are 'clean' and more easily understood by all stakeholders.

Also, because processes are able to be expressed as a parent process with a number of supporting worklets/exlets, the entire process is much easier to comprehend (i.e. through the lens of its composite parts) than is the case when all the possible flows and branches are encapsulated into the one monolithic model. A worker, with a full understanding of their own work processes, can more easily see that work in a discrete worklet specification, than if it is shown to them as a small part of a much larger, more complex specification.

Therefore, the worklet service is able to represent a workflow specification at different levels of granularity to suit all stakeholders.

The Elevation of Exceptions to 'First-Class Citizens' The worklet service has been built around the idea that exceptions, or deviations from the process specification, are a natural occurrence of almost every instantiation of the process, and give rise to learning experiences. Thus the service provides a fully featured exception handling sub-service that detects (through constraint checking), reacts to, handles and incorporates exceptions and the way they are handled as they occur. The exceptions are handled on-system so that the organisational memory of how they are handled is retained.

The service also allows for unexpected exceptions to be handled during execution, so that a process instance needn't 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 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.

A primary feature of the worklet service is that it has been designed and implemented as a discrete service, and so offers all of its benefits to a wide range of workflow management systems, allowing them to fully 'worklet-ise' their otherwise static processes. The worklet service:

- Keeps the parent model clean and relatively simple;
- Promotes the reuse of sub-processes in different models;
- Allows standard processes to be used as exception handling compensation processes, and vice versa;
- Maintains an extensible repertoire of actions 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 for certain contexts automatically;

- Maintains a repertoire of fully encapsulated, discrete worklets that allow for easier verification and modification;
- Allows a model to evolve without the need to stop and modify the design of the whole specification when an exception occurs;
- By de-coupling the monolithic process model, models can be built that vary from loosely to tightly defined and so supports late binding of processes; and
- Allows a model to be considered from many levels of granularity.

There are a number of further research topic possibilities that arise from this work, such as: deeper empirical studies comparing the worklet approach to classic workflow approaches and measuring the benefits of each in terms of the criteria above; porting the worklet service to other workflow systems (for example, IBM Websphere and/or Oracle BPEL); exploring the advantages of mixing different modelling styles and approaches, leading to recommendations of in what circumstances the various approaches are best used; and stronger support for process mining analysis using both the process logs generated by the service and the structure, content and evolution of the various ripple-down rule sets of specifications.

In summary, through a combination of the framework it is built on and the mechanisms available through both its selection and exception handling sub-services, the worklet service offers a wide-ranging solution to the issues of flexibility and exception handling in workflow systems. In fact, the benefits offered through each sub-service can be combined to deliver a far-reaching set of capabilities that would serve the needs of a wide variety of work environments and processes, from the rigid to the creative.

Appendix A

CPN Declarations

This appendix lists the declarations for the coloured petri nets shown and discussed in Chapter 5.

```
(* Standard declarations *)
     colset UNIT = unit;
     colset INT = int;
     colset BOOL = bool;
     colset STRING = string;
     colset ID_STR = string;

(* Engine-side declarations *)
     colset SPECID = ID_STR;
     colset CASEID = ID_STR;
     colset TASKID = ID_STR;
     colset ITEMID = ID_STR;
     colset PARAM = product STRING * STRING;
     colset PARAMS = list PARAM;
     colset DATALIST = PARAMS;
     colset WIR = product SPECID * CASEID * TASKID * ITEMID * DATALIST;
     colset WIRLIST = list WIR;
     colset CASExDATA = product CASEID * DATALIST;
     colset CASExITEM = product CASEID * ITEMID;
     colset CASExID = product CASEID * ID_STR;
     colset CASExSPEC = product CASEID * SPECID;
     colset METHOD = with restartWorkItem | cancelWorkItem | forceCompleteWorkItem |
                          suspendWorkItem | unsuspendWorkItem | cancelCase |
                          getCases | getListOfLiveWorkItems | launchCase |
                          updateWorkItemData | updateCaseData |
                          checkIn | checkOut | getInputParams;
     colset RESPONSE = with rspLaunchCase | rspCheckOut | rspCheckIn | rspGetCases |
                            rspUnsuspendWorkItem | rspGetListOfLiveWorkItems |
                            rspCancelCase | rspSuspendWorkItem | rspGetInputParams;
     colset POST = product METHOD * PARAMS;
     colset RESP = product RESPONSE * PARAMS;

(* Service-side Declarations *)
     colset EXTYPE = with CasePreConstraint | CasePostConstraint |
                          ItemPreConstraint | ItemPostConstraint |
                          ConstraintViolation | ItemAbort | ResourceUnavailable |
                          TimeOut | CaseExternal | ItemExternal | Selection;
```

```
    colset EXLEVEL = with exCase | exItem ;
    colset CSTONE = PARAMS;
    colset COND = STRING;
    colset ACTION = with restart | fail | complete |
                        suspend | continue | remove |
                        compensate | select;
    colset TARGET = STRING;
    colset WORKLET = TARGET;
    colset PRIMITIVE = product ACTION * TARGET ;
    colset EXLET = list PRIMITIVE;
    colset NODE_ID = INT;
    colset CHILD_NODE_ID = NODE_ID;
    colset RNODE = product NODE_ID * COND * EXLET * CSTONE * CHILD_NODE_ID * CHILD_NODE_ID;
    colset TREE = list RNODE;
    colset TREEID = product SPECID * TASKID * EXTYPE;
    colset IDLIST = list ID_STR;
    colset IDxEXLET = product ID_STR * EXLET;
    colset IDxPRIMITIVE = product ID_STR * PRIMITIVE;
    colset IDxTARGET = product ID_STR * TARGET;
    colset IDxEXLEVEL = product ID_STR * EXLEVEL;
    colset IDxDATA = product ITEMID * DATALIST;
    colset WORKLETxDATA = product WORKLET * DATALIST;
    colset IDxTREExDATA = product ID_STR * TREE * DATALIST;
    colset IDxTREEIDxDATA = product ID_STR * TREEID * DATALIST;
    colset IDxEXLETxDATA = product ID_STR * EXLET * DATALIST;
    colset TREEPARAMS = product EXLEVEL * ID_STR * TREEID * DATALIST;
    colset IDxIDLIST = product ID_STR * IDLIST;
    colset evCASECONSTRAINT = product SPECID * CASEID * DATALIST * BOOL;
    colset evITEMCONSTRAINT = product WIR * DATALIST * BOOL;
    colset TREEMAP = product TREEID * TREE;
    colset TREELIST = list TREEMAP;
    colset ITEMxDATA = product ITEMID * DATALIST;
    colset WIRxDATA = product WIR * DATALIST;
    colset RNODExEVAL = product RNODE * BOOL ;
    colset TRIGGER = STRING;
    colset evEXTERNAL = product SPECID * CASEID * TASKID * ITEMID * DATALIST * TRIGGER;

(* Variables *)
    var pre : BOOL ;
    var c, wc : CASEID;
    var nid: CHILD_NODE_ID;
    var d, cd, wd : DATALIST;
    var xlt : EXLET;
    var xlv : EXLEVEL;
    var id : ID_STR;
    var idl : IDLIST;
    var n : INT;
    var i : ITEMID;
    var p : PARAM;
    var pl : PARAMS;
    var postAck : POST;
    var pr : PRIMITIVE;
    var node, root, prevTrue : RNODE;
    var s : SPECID;
    var tar : TARGET;
    var t : TASKID;
    var tree : TREE;
    var tid : TREEID;
```

```
    var tList : TREELIST;
    var trig : TRIGGER;
    var e: UNIT;
    var wir : WIR;
    var wirList : WIRLIST;
    var wkt : WORKLET ;

(* Functions *)
    fun inPList(p:PARAM, []:PARAMS) = false |
        inPList(p, h::pl) = if #1p = #1h then true else inPList(p, pl);

    fun repl(p:PARAM, []:PARAMS) = [] |
        repl(p, h::pl) = if #1p = #1h then p::pl else h::repl(p, pl);

    fun upData([]:PARAMS, q:PARAMS) = q |
        upData(p::pl, q) = repl(p, upData(pl, q));

    fun PtoStr(p:PARAM) = "(" ^ #1 p ^ "," ^ #2 p ^ ")";

    fun PLtoS([]:PARAMS) = "" |
        PLtoS(p::pl) = PtoStr(p) ^ PLtoS(pl);

    fun getNode(id:NODE_ID, []:TREE) = [] |
        getNode(id, n::t) = if #1n = id then [n] else getNode(id, t);

    fun getCond(n:RNODE) = #2n;

    fun getExlet(n:RNODE) = #3n;

    fun getChildID(n:RNODE, t:BOOL) = if t then #5n else #6n;

    fun getAction(p:PRIMITIVE) = #1p;

    fun getTarget(p:PRIMITIVE) = #2p;

    fun getWorklet(p:PRIMITIVE) = getTarget(p);

    fun getPrim(i:INT, []:EXLET) = [] |
        getPrim(i, x::xl) = if i=0 then [x] else getPrim(i-1, xl);

    fun getIDL([]:PARAMS) = [] |
        getIDL(p::pl) = #2p::getIDL(pl);

    fun getID(p:PARAM) = #2p;

    fun eval(c:COND, d:DATALIST) = if c="true" then true else false;

    fun deList([]) = empty | deList(i::il) = 1`i ++ deList(il);

    fun getWRsp(m:METHOD) = if m=checkIn then rspCheckIn
        else if m=checkOut then rspCheckOut else rspUnsuspendWorkItem;

    fun hasTree(tid:TREEID, []:TREELIST) = false |
        hasTree(tid, tm::tl) = if tid = #1tm then true else hasTree(tid, tl);

    fun getTree(tid:TREEID, []:TREELIST) = [] |
        getTree(tid, tm::tl) = if tid = #1tm then #2tm else getTree(tid, tl);
```

Appendix B

Worklet Selection Sequence Diagrams

This series of sequence diagrams describe the selection processes of the worklet service.

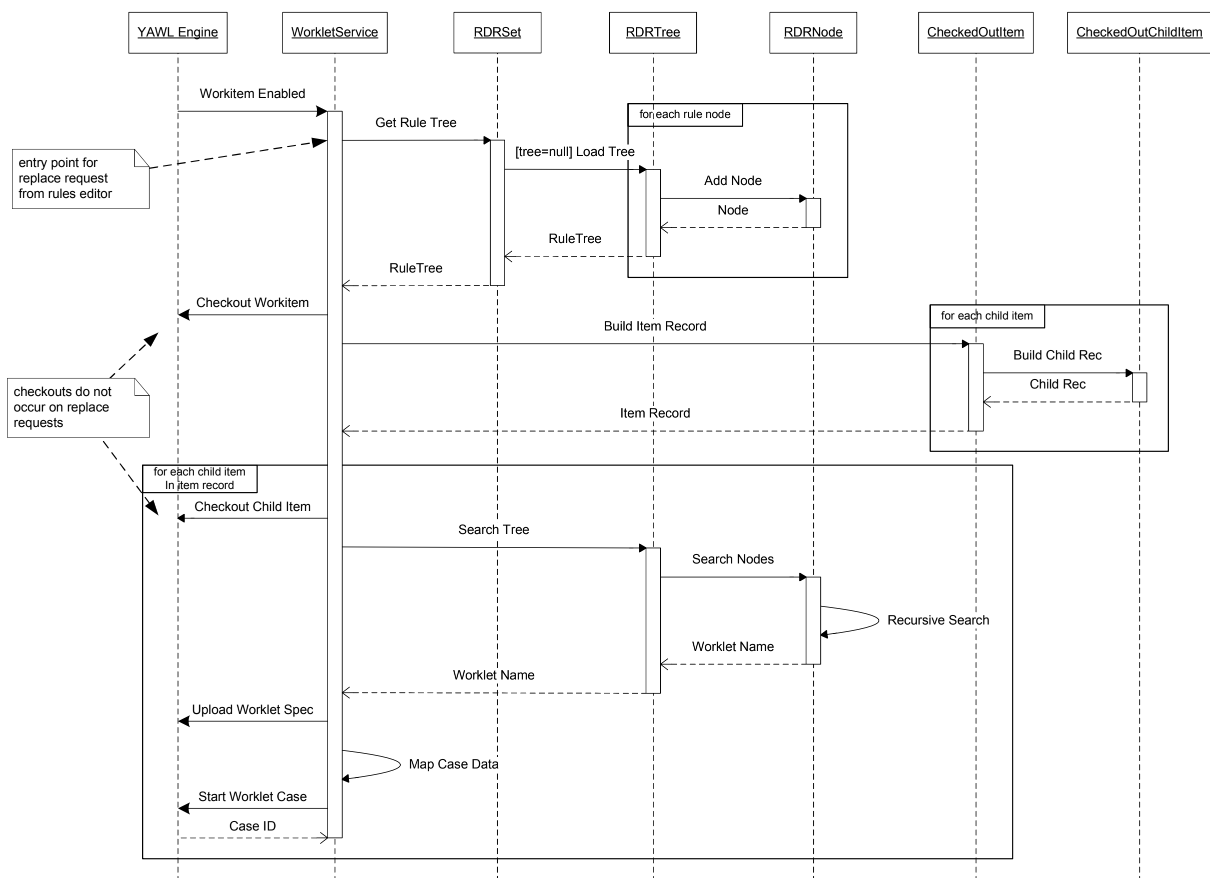

Figure B.1: Sequence Diagram for Workitem Substitution

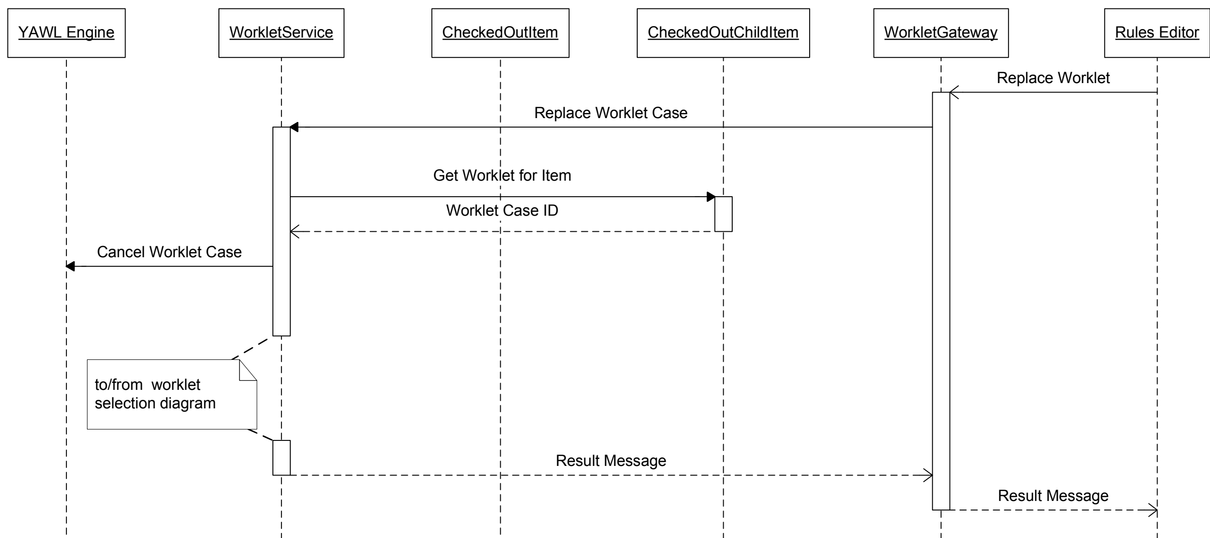

Figure B.2: Sequence Diagram for Worklet Replacement

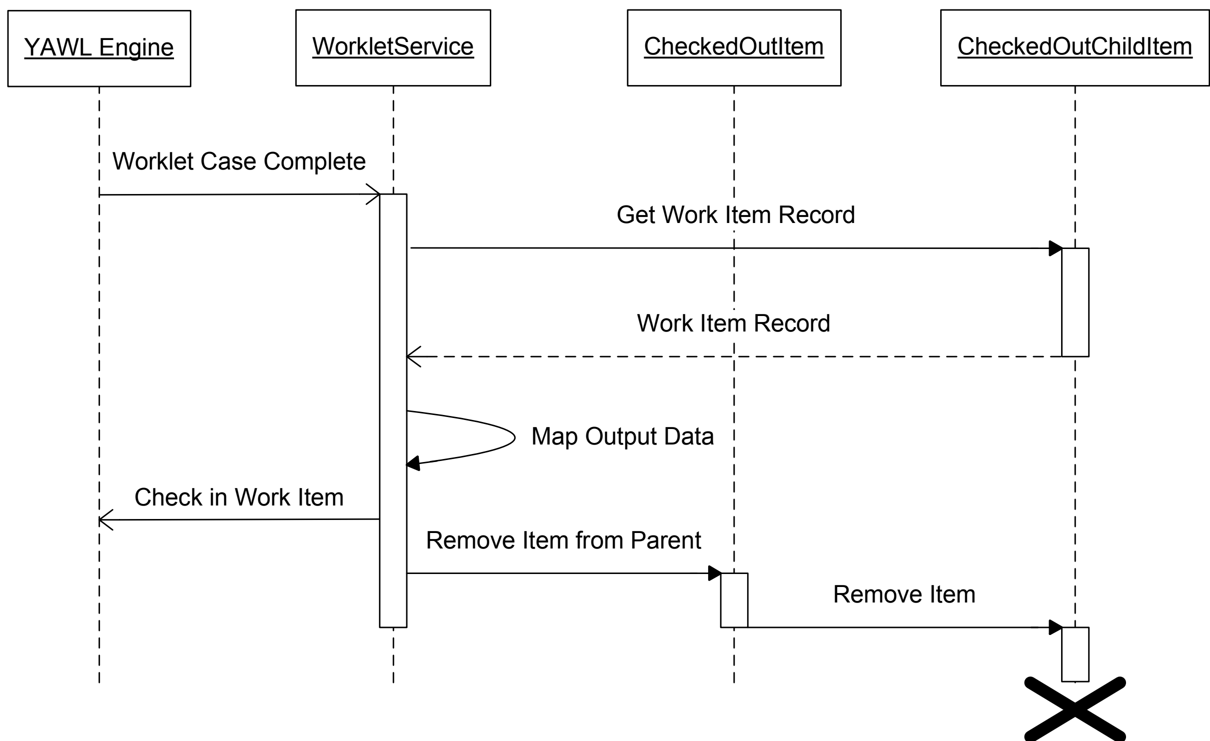

Figure B.3: Sequence Diagram for Worklet Completion

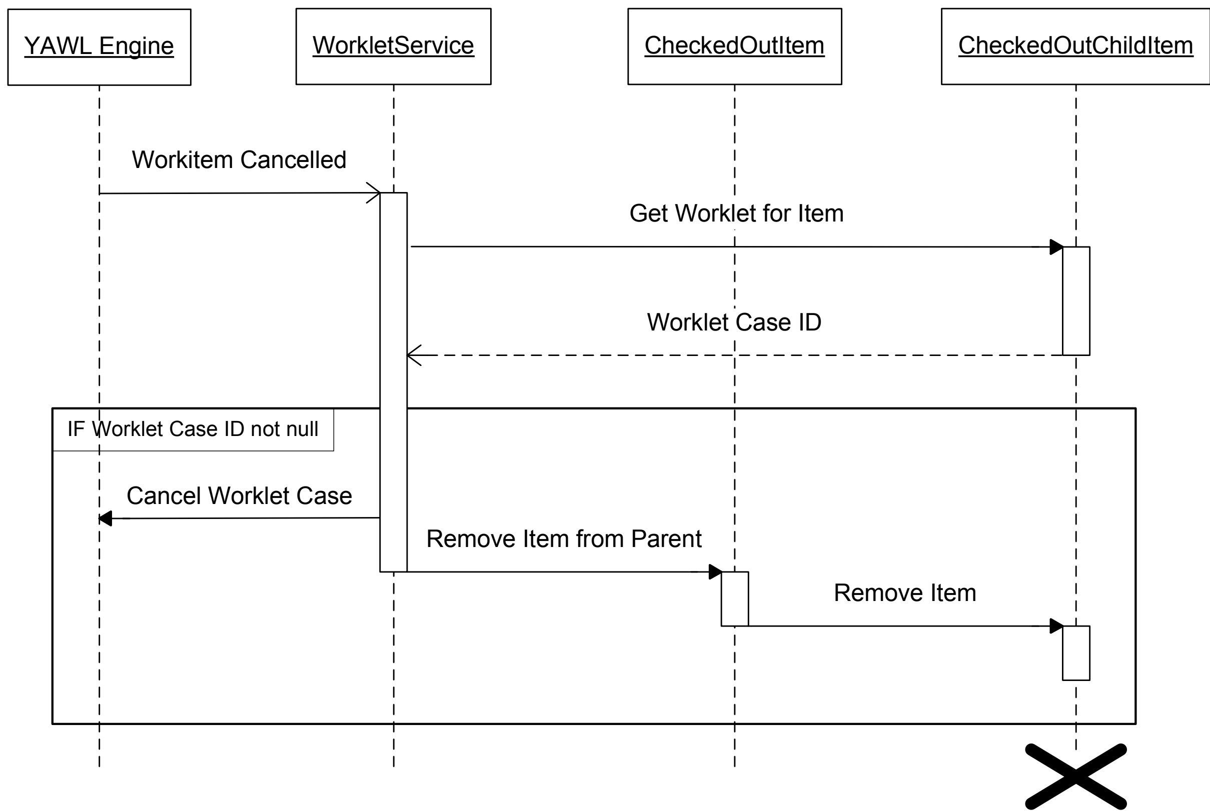

Figure B.4: Sequence Diagram for Workitem Cancellation

Appendix C

Sample Audit Log File Output

The two samples in this appendix are extracted from the worklet service log file generated while the service is in operation. The worklet service writes extensive information to this log file to provide detailed descriptions of its handling of each and every process instance.

The logging operation is achieved through the use of the open source *log4j* package.

C.1 Selection Service Sample

```
2006-12-15 11:37:11,828 [INFO ] WorkletService       :- HANDLE ENABLED WORKITEM EVENT
2006-12-15 11:37:11,843 [INFO ] WorkletService       :- Connection to engine is active
2006-12-15 11:37:11,984 [INFO ] WorkletService       :- Received workitem for worklet substitution: 100000:3_Treat
2006-12-15 11:37:11,984 [INFO ] WorkletService       :-    specId = Casualty_Treatment
2006-12-15 11:37:12,000 [INFO ] WorkletService       :- Ruleset found for workitem: 100000:3_Treat
2006-12-15 11:37:12,015 [INFO ] WorkletService       :- Checking parent workitem out of engine: 100000:3_Treat
2006-12-15 11:37:12,625 [INFO ] WorkletService       :-    checkout successful: 100000:3_Treat
2006-12-15 11:37:12,625 [INFO ] WorkletService       :- Checking out child workitems...
2006-12-15 11:37:12,625 [INFO ] WorkletService       :-    child already checked out with parent: 100000.3:3_Treat
2006-12-15 11:37:12,656 [INFO ] WorkletService       :- Processing worklet substitution for workitem: 100000.3:3_Treat
2006-12-15 11:37:12,656 [INFO ] WorkletService       :- Rule search returned worklet(s): TreatFever
2006-12-15 11:37:13,062 [INFO ] WorkletService       :- Successfully uploaded worklet specification: TreatFever
2006-12-15 11:37:13,250 [INFO ] WorkletService       :- Launched case for worklet TreatFever with ID: 100001
2006-12-15 11:38:25,781 [INFO ] WorkletService       :- HANDLE COMPLETE CASE EVENT
2006-12-15 11:38:25,781 [INFO ] WorkletService       :- ID of completed case: 100001
2006-12-15 11:38:25,781 [INFO ] WorkletService       :- Connection to engine is active
2006-12-15 11:38:25,781 [INFO ] WorkletService       :- Workitem this worklet case ran in place of is: 100000.3:3_Treat
2006-12-15 11:38:25,828 [INFO ] WorkletService       :- Removed from cases started: 100001
2006-12-15 11:38:25,828 [INFO ] WorkletService       :- Handling of workitem completed - checking it back in to engine
2006-12-15 11:38:26,328 [INFO ] WorkletService       :- Successful checkin of work item: 100000.3:3_Treat
2006-12-15 11:38:26,328 [INFO ] WorkletService       :- Removed from handled child workitems: 100000.3:3_Treat
2006-12-15 11:38:26,328 [INFO ] WorkletService       :- No more child cases running for workitem: 100000:3_Treat
2006-12-15 11:38:26,343 [INFO ] WorkletService       :- Completed handling of workitem: 100000:3_Treat
```

C.2 Exception Service Sample

```
2006-09-12 12:16:31,875 [INFO ] ExceptionService     :- HANDLE CHECK CASE CONSTRAINT EVENT
2006-09-12 12:16:31,984 [INFO ] ExceptionService     :- Checking constraints for start of case 20 (of specification: OrganiseConcert)
2006-09-12 12:16:32,093 [INFO ] ExceptionService     :- No pre-case constraints defined for spec: OrganiseConcert
2006-09-12 12:16:32,109 [INFO ] ExceptionService     :- HANDLE CHECK WORKITEM CONSTRAINT EVENT
2006-09-12 12:16:32,156 [INFO ] ExceptionService     :- Checking pre-constraints for workitem: 20:BookStadium_5
2006-09-12 12:16:32,281 [INFO ] ExceptionService     :- No pre-task constraints defined for task: BookStadium
2006-09-12 12:28:17,968 [INFO ] ExceptionService     :- HANDLE CHECK WORKITEM CONSTRAINT EVENT
2006-09-12 12:28:18,000 [INFO ] ExceptionService     :- Checking pre-constraints for workitem: 20:SellTickets_3
2006-09-12 12:28:18,015 [INFO ] ExceptionService     :- No pre-task constraints defined for task: SellTickets
2006-09-12 12:28:18,078 [INFO ] ExceptionService     :- HANDLE CHECK WORKITEM CONSTRAINT EVENT
2006-09-12 12:28:18,093 [INFO ] ExceptionService     :- Checking post-constraints for workitem: 20.1:BookStadium_5
2006-09-12 12:28:18,093 [INFO ] ExceptionService     :- No post-task constraints defined for task: BookStadium
2006-09-12 12:56:08,000 [INFO ] ExceptionService     :- HANDLE CHECK WORKITEM CONSTRAINT EVENT
2006-09-12 12:56:08,015 [INFO ] ExceptionService     :- Checking pre-constraints for workitem: 20:DoShow_4
2006-09-12 12:56:08,140 [INFO ] ExceptionService     :- Workitem 20:DoShow_4 failed pre-task constraints
2006-09-12 12:56:08,140 [INFO ] ExceptionService     :- Invoking exception handling process for item: 20:DoShow_4
2006-09-12 12:56:08,156 [INFO ] ExceptionService     :- Exception process step 1. Action = suspend, Target = workitem
2006-09-12 12:56:08,171 [INFO ] ExceptionService     :- Successful work item suspend: 20:DoShow_4
2006-09-12 12:56:08,203 [INFO ] ExceptionService     :- Exception process step 2. Action = compensate, Target = ChangeToMidVenue
2006-09-12 12:56:08,343 [INFO ] WorkletService       :- Worklet specification 'ChangeToMidVenue' is already loaded in Engine
2006-09-12 12:56:08,546 [INFO ] WorkletService       :- Launched case for worklet ChangeToMidVenue with ID: 21
2006-09-12 12:56:08,578 [INFO ] ExceptionService     :- HANDLE CHECK WORKITEM CONSTRAINT EVENT
2006-09-12 12:56:08,593 [INFO ] ExceptionService     :- Checking post-constraints for workitem: 20.2:SellTickets_3
2006-09-12 12:56:08,593 [INFO ] ExceptionService     :- No post-task constraints defined for task: SellTickets
2006-09-12 12:56:08,593 [INFO ] ExceptionService     :- HANDLE CHECK CASE CONSTRAINT EVENT
2006-09-12 12:56:08,593 [INFO ] ExceptionService     :- Checking constraints for start of case 21 (of specification: ChangeToMidVenue)
2006-09-12 12:56:08,609 [INFO ] ExceptionService     :- No pre-case constraints defined for spec: ChangeToMidVenue
2006-09-12 12:56:08,609 [INFO ] ExceptionService     :- HANDLE CHECK WORKITEM CONSTRAINT EVENT
2006-09-12 12:56:08,640 [INFO ] ExceptionService     :- Checking pre-constraints for workitem: 21:CancelStadium_3
2006-09-12 12:56:08,656 [INFO ] ExceptionService     :- No pre-task constraints defined for task: CancelStadium
2006-09-12 13:02:48,171 [INFO ] ExceptionService     :- HANDLE CHECK WORKITEM CONSTRAINT EVENT
2006-09-12 13:02:48,187 [INFO ] ExceptionService     :- Checking pre-constraints for workitem: 21:Book_Ent_Centre_5
2006-09-12 13:02:48,234 [INFO ] ExceptionService     :- No pre-task constraints defined for task: Book_Ent_Centre
2006-09-12 13:02:48,250 [INFO ] ExceptionService     :- HANDLE CHECK WORKITEM CONSTRAINT EVENT
2006-09-12 13:02:48,265 [INFO ] ExceptionService     :- Checking post-constraints for workitem: 21.1:CancelStadium_3
2006-09-12 13:02:48,265 [INFO ] ExceptionService     :- No post-task constraints defined for task: CancelStadium
2006-09-12 13:10:10,468 [INFO ] ExceptionService     :- HANDLE CHECK WORKITEM CONSTRAINT EVENT
2006-09-12 13:10:10,484 [INFO ] ExceptionService     :- Checking pre-constraints for workitem: 21:Tell_Punters_4
2006-09-12 13:10:10,500 [INFO ] ExceptionService     :- No pre-task constraints defined for task: Tell_Punters
2006-09-12 13:10:10,500 [INFO ] ExceptionService     :- HANDLE CHECK WORKITEM CONSTRAINT EVENT
2006-09-12 13:10:10,515 [INFO ] ExceptionService     :- Checking post-constraints for workitem: 21.2:Book_Ent_Centre_5
2006-09-12 13:10:10,515 [INFO ] ExceptionService     :- No post-task constraints defined for task: Book_Ent_Centre
2006-09-12 13:13:59,281 [INFO ] ExceptionService     :- HANDLE CHECK CASE CONSTRAINT EVENT
2006-09-12 13:13:59,281 [INFO ] ExceptionService     :- Checking constraints for end of case 21
2006-09-12 13:13:59,281 [INFO ] ExceptionService     :- No post-case constraints defined for spec: ChangeToMidVenue
2006-09-12 13:13:59,296 [INFO ] ExceptionService     :- Worklet ran as exception handler for case: 20
2006-09-12 13:13:59,437 [INFO ] ExceptionService     :- Exception process step 3. Action = continue, Target = workitem
2006-09-12 13:13:59,468 [INFO ] ExceptionService     :- Successful work item unsuspend: 20:DoShow_4
2006-09-12 13:13:59,515 [INFO ] ExceptionService     :- HANDLE CHECK WORKITEM CONSTRAINT EVENT
2006-09-12 13:13:59,531 [INFO ] ExceptionService     :- Checking post-constraints for workitem: 21.3:Tell_Punters_4
2006-09-12 13:13:59,531 [INFO ] ExceptionService     :- No post-task constraints defined for task: Tell_Punters
2006-09-12 13:13:59,546 [INFO ] ExceptionService     :- Exception monitoring complete for case 21
2006-09-12 13:13:59,750 [INFO ] ExceptionService     :- HANDLE CHECK CASE CONSTRAINT EVENT
2006-09-12 13:13:59,875 [INFO ] ExceptionService     :- Checking constraints for end of case 20
2006-09-12 13:13:59,953 [INFO ] ExceptionService     :- No post-case constraints defined for spec: OrganiseConcert
2006-09-12 13:14:00,046 [INFO ] ExceptionService     :- HANDLE CHECK WORKITEM CONSTRAINT EVENT
2006-09-12 13:14:00,046 [INFO ] ExceptionService     :- Checking post-constraints for workitem: 20.3:DoShow_4
2006-09-12 13:14:00,156 [INFO ] ExceptionService     :- No post-task constraints defined for task: DoShow
2006-09-12 13:14:00,171 [INFO ] ExceptionService     :- Exception monitoring complete for case 20
```

Bibliography

[1] Wil van der Aalst. Generic workflow models: How to handle dynamic change and capture management information? In M. Lenzerini and U. Dayal, editors, *Proceedings of the Fourth IFCIS International Conference on Cooperative Information Systems (CoopIS'99)*, pages 115–126, Edinburgh, Scotland, 1999. IEEE Computer Society Press. 35

[2] Wil van der Aalst and Kees van Hee. *Workflow Management: Models, Methods and Systems*. The MIT Press, Cambridge, Massachusetts, New Ed edition, 2004. 1, 39

[3] W.M.P. van der Aalst. Exterminating the dynamic change bug: A concrete approach to support workflow change. *Information Systems Frontiers*, 3(3):297–317, 2001. 67

[4] 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 *Lecture Notes in Computer Science*, pages 142–159, Riga, Latvia, June 2004. Springer Verlag. 91, 111, 112, 114, 117, 118

[5] W.M.P. van der Aalst, P. Barthelmess, C.A. Ellis, and J. Wainer. Proclets: A framework for lightweight interacting workflow processes. *International Journal of Cooperative Information Systems*, 10(4):443–482, 2001. 33

[6] W.M.P. van der Aalst and T. Basten. Inheritance of workflows: An approach to tackling problems related to change. *Theoretical Computer Science*, 270((1-2)):125–203, 2002. 67

[7] W.M.P. van der Aalst and P.J.S. Berens. Beyond workflow management: Product-driven case handling. In S. Ellis, T. Rodden, and I. Zigurs, editors, *Proceedings of the Inter-*

national ACM SIGGROUP Conference on Supporting Group Work, pages 42–51, New York, 2001. ACM Press. 8, 40, 45

[8] W.M.P. van der Aalst and B.F. van Dongen. Discovering Workflow Performance Models from Timed Logs. In *Proceedings of the International Conference on Engineering and Deployment of Cooperative Information Systems (EDCIS 2002)*, volume 2480 of *Lecture Notes in Computer Science*, pages 45–63, Beijing, China, 2002. Springer. 39, 59

[9] 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. 32, 117

[10] W.M.P. van der Aalst and S. Jablonski. Dealing with workflow change: Identification of issues and solutions. *International Journal of Computer Systems, Science, and Engineering*, 15(5):267–276, 2000. 31

[11] W.M.P. van der Aalst and M. Pesic. DecSerFlow: Towards a truly declarative service flow language. In M. Bravetti, M. Nunez, and G. Zavattaro, editors, *Proceedings of the International Conference on Web Services and Formal Methods (WS-FM 2006)*, volume 4184 of *Lecture Notes in Computer Science*, pages 1–23, Vienna, Austria, September 2006. Springer-Verlag, Berlin, Germany. 28

[12] W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005. 21, 91, 112, 117

[13] W.M.P. van der Aalst, B.F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A.J.M.M. Weijters. Workflow mining: A survey of issues and approaches. *Data and Knowledge Engineering*, 47(2):237–267, 2003. 88

[14] W.M.P. van der Aalst, A.J.M.M. Weijters, and L. Maruster. Workflow mining: Which processes can be rediscovered? BETA Working Paper Series, WP 74, Eindhoven University of Technology, Eindhoven, The Netherlands, 2002. 39, 59, 62

[15] W.M.P. van der Aalst, T. Weijters, and L. Maruster. Workflow mining: discovering process models from event logs. *Knowledge and Data Engineering*, 16(9):1128–1142, 2004. 59

[16] 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. 7, 26, 40

[17] 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, Seattle, Washington, USA, 1998. ACM Press. 4, 8, 32, 55

[18] A. Agostini, G. De Michelis, M.A. Grasso, and S. Patriarca. Reengineering a business process with an innovative workflow management system: a case study. In Simon Kaplan, editor, *Proceedings of the Conference on Organisational Computing Systems*, pages 154–165, Milpitas, California, USA, November 1993. ACM. 30

[19] Alessandra Agostini and Giorgio De Michelis. Modeling the document flow within a cooperative process as a resource for action. *Technical Report CTL-DSI, University of Milano, Milano, Italy*, 1996. 30, 59

[20] Alessandra Agostini and Giorgio De Michelis. A light workflow management system using simple process models. *Computer Supported Cooperative Work*, 9(3/4):335–363, 2000. 35

[21] Alessandra Agostini, Giorgio De Michelis, and Katia Petruni. Keeping workflow models as simple as possible. In *Proceedings of the Workshop on Computer-Supported Cooperative Work, Petri-Nets and related formalisms, within the 15th International Conference on Application and Theory of Petri Nets (ATPN '94)*, pages 11–23, Zaragoza, Spain, June 1994. Springer. 35

[22] Rakesh Agrawal, Dimitrios Gunopulos, and Frank Leymann. Mining process models from workflow logs. In *Proceedings of the 6th International Conference on Extending Database Technology*, volume 1377 of *Lecture Notes in Computer Science*, pages 469–484, Valencia, Spain, 1998. 39, 59

[23] L. Aldred, W. M.P. van der Aalst, M. Dumas, and A.H.M. ter Hofstede. On the notion of coupling in communication middleware. In Robert Meersman and Zahir Tari, editors, *Proceedings of the International Symposium on Distributed Objects and Applications (DOA '05)*, volume 3761 of *Lecture Notes in Computer Science*, pages 1015–1033, Agia Napa, Cyprus, November 2005. Springer Verlag. 92

[24] Jakob E. Bardram. I love the system - I just don't use it! In *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work (GROUP'97)*, pages 251–260, Phoenix, Arizona, USA, November 1997. ACM. 6, 29, 32, 51, 55, 57, 59, 64

[25] Jakob E. Bardram. Plans as situated action: an Activity Theory approach to workflow systems. In *Proceedings of the Fifth European Conference on Computer Supported Cooperative Work (ECSCW'97)*, pages 17–32, Lancaster, United Kingdom, 1997. Kluwer. 2, 4, 58

[26] Jakob E. Bardram. *Collaboration, Coordination and Computer Support*. PhD thesis, University of Aarhus, Denmark, 1998. 64

[27] Jakob E. Bardram. Designing for the dynamics of cooperative work activities. In *Proceedings of the ACM 1998 Conference on Computer-Supported Cooperative Work*, pages 89–98, Seattle, Washington, USA, November 1998. ACM. 53

[28] L. Baresi, F. Casati, S. Castano, M. Fugini, I. Mirbel, and B. Pernici. WIDE workflow development methodology. In *Proceedings of the International Joint Conference on Work Activities Coordination and Collaboration (WACC '99)*, pages 19–28, San Francisco, California, USA, 1999. ACM. 34, 37

[29] P. Barthelmess and K.M. Anderson. A view of software development environments based on activity theory. *Computer Supported Cooperative Work*, 11(1-2):13–37, 2002. Special issue on Activity Theory. 53

[30] P. Barthelmess and J. Wainer. Workflow systems: a few definitions and a few suggestions. In *Proceedings of the ACM Conference on Organizational Computing Systems (COOCS'95)*, pages 138–147, Milpitas, California, USA, 1995. ACM. 4, 5, 33

[31] Len Bass, Paul Clement, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, Reading, 1998. 14, 60

[32] Paul Berens. *The FLOWer Case Handling Approach: Beyond Workflow Managment*, chapter 15, pages 363–395. In Dumas et al. [66], 2005. 26, 40, 59

[33] P. Bichler, G. Preuner, and M. Schrefl. Workflow transparency. In Antoni Olivé and Joan Antoni Pastor, editors, *Proceedings of the 9th International Conference on Advanced Information Systems Engineering (CAiSE'97)*, volume 1250 of *Lecture Notes*

in Computer Science, pages 423–436, Barcelona, Catalonia, Spain, June 1997. Springer. 24

[34] 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. 1, 7

[35] I. Bider and M. Khomyakov. What is wrong with workflow management systems or is it possible to make them flexible? In *Proceedings of the 6th International Workshop on Groupware (CRIWG2000)*, pages 138–141, Madeira Island, Portugal, October 2000. IEEE Computer Society Press. 33

[36] Richard Boardman. Activity-centric social navigation. In Cecile Paris, Nadine Ozkan, Steve Howard, and Shijian Lu, editors, *Companion of OZCHI 2000 Conference*. The University of Technology, Sydney, Australia, CSIRO, 4-8 December 2000. 38

[37] Susanne Bødker. Activity theory as a challenge to systems design. In Hans-Erik Nissen, editor, *The Information Systems Research Arena of the 90s, Challenges, Perceptions and Alternative Approaches*, Amsterdam, 1991. North Holland. 46, 64

[38] Susanne Bødker and Joan Greenbaum. Design of information systems: Things versus people. In Eileen Green, Jenny Owen, and Den Pain, editors, *Gendered by Design?: Information Technology and Office Systems*, chapter 3, pages 53–63. Taylor and Francis, London, 1993. 46, 64

[39] Alex Borgida and Takahiro Murata. A unified framework for exceptions in workflow and process models - an approach based on persistent objects. Technical report, Rutgers University, Newark, New Jersey, USA, November 1998. 34, 37

[40] 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, California, USA, February 1999. ACM Press. 2, 5, 37, 45

[41] Alex Borgida and Takahiro Murata. Workflow as persistent objects with persistent exceptions: a framework for flexibility. *SIGGROUP Bulletin*, 20(3):4–5, 1999. 34

[42] Paolo Bouquet, Chiara Ghidini, Fausto Giunchiglia, and Enrico Blanzieri. Theories and uses of context in knowledge representation and reasoning. *Journal of Pragmatics*, 35(3):455–484, 2003. 68, 69

[43] J. Cardoso, Z. Luo, J. Miller, A. Sheth, and K. Kochut. Survivability architecture for workflow systems. In *Proceedings of the 39th Annual ACM Southeast Conference (ACM-SE'01)*, pages 207–214, Athens, Georgia, USA, March 2001. ACM. 39

[44] S. Carlsen, J. Krogstie, A. Slvberg, and O. Lindland. Evaluating flexible workflow systems. In *Proceedings of the 30th Hawaii International Conference on System Sciences (HICSS-30)*, pages 230–239, Maui, Hawaii, January 1997. IEEE Computer Society. 10

[45] Steinar Carlsen. Comprehensible business process models for process improvement and process support. In *Proceedings of the 3rd Doctoral Consortium on Advanced Information Systems Engineering*, Heraklion, Greece, May 1996. Published online: `http://www.softlab.ntua.gr/~bxb/caise96dc/caise96dc/caise96dc/Papers/carlsen.html`. Access 23 May, 2002. 13, 61

[46] F. Casati, P. Grefen, B. Pernici, G. Pozzi, and G. Sanchez. WIDE workflow model and architecture. Technical report 96-19, Centre for Telematics and Information Technology (CTIT), University of Twente, The Netherlands, 1996. 34, 36

[47] Fabio Casati. A discussion on approaches to handling exceptions in workflows. In *Proceedings of the CSCW Workshop on Adaptive Workflow Systems*, Seattle, USA, November 1998. 1, 2, 5, 27, 36, 37, 57

[48] Fabio Casati, Silvana Castano, Maria Grazia Fugini, Isabelle Mirbel, and Barbara Pernici. Werde: A pattern-based tool for exception design in workflows. In *Proceedings of the Italian Symposium on Advanced Database Systems (SEBD)*, pages 237–252, Ancona, Italy, 1998. 37, 57

[49] Fabio Casati, Stefano Ceri, Stefano Paraboschi, and Giuseppe Pozzi. Specification and implementation of exceptions in workflow management systems. *ACM Transactions on Database Systems*, 24(3):405–451, 1999. 6, 32, 34, 35

[50] Fabio Casati, MariaGrazia Fugini, and Isabelle Mirbel. An environment for designing exceptions in workflows. *Information Systems*, 24(3):255–273, 1999. 6, 37

[51] Fabio Casati, Ski Ilnicki, LiJie Jin, Vasudev Krishnamoorthy, and Ming-Chien Shan. Adaptive and dynamic composition in eFlow. In *Proceedings of the 12th International Conference on Advanced Information Systems Engineering (CAiSE 2000)*, volume 1789 of *Lecture Notes in Computer Science*, pages 13–31, Stockholm, Sweden, 2000. Springer. 26, 34, 37, 63

[52] Fabio Casati and Giuseppe Pozzi. Modelling exceptional behaviours in commercial workflow management systems. In *Proceedings of the 4th IFCIS International Conference on Cooperative Information Systems (CoopIS'99)*, pages 127–138, Edinburgh, Scotland, 1999. IEEE. 23, 24, 32, 62

[53] Stefania Castellani and François Pacull. XFolders: A flexible workflow system based on electronic circulation folders. In *Proceedings of the 13th International Workshop on Database and Expert Systems Applications (DEXA '02)*, pages 307–312, Washington, USA, 2002. IEEE Computer Society. 28

[54] Dickson Chiu, Kamalakar Karlapalem, and Qing Li. Exception handling with workflow evolution in ADOME-WFMS: a taxonomy and resolution techniques. *SIGGROUP Bulletin*, 20(3):8, 1999. 32, 37, 57

[55] Dickson Chiu, Qing Li, and Kamalakar Karlapalem. A meta modeling approach to workflow management systems supporting exception handling. *Information Systems*, 24(2):159–184, 1999. 37, 57

[56] Dickson Chiu, Qing Li, and Kamalakar Karlapalem. A logical framework for exception handling in ADOME workflow management system. In *Proceedings of the 12th International Conference on Advanced Information Systems Engineering (CAiSE 2000)*, volume 1789 of *Lecture Notes in Computer Science*, pages 110–125, Stockholm, Sweden, 2000. Springer. 27, 37, 57

[57] Patricia Collins, Shilpa Shukla, and Davide Redmiles. Activity Theory and system design: A view from the trenches. *Computer Supported Cooperative Work*, 11(1-2):55–80, 2002. 29, 50, 64

[58] 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 Intelli-*

gence Conference, volume 406 of *Lecture Notes in Artificial Intelligence*, pages 292–306, Adelaide, Australia, November 1988. Springer-Verlag. 69, 71

[59] COSA BPM product description. http://www.cosa-bpm.com/project/docs/COSA_BPM_5_Productdescription_eng.pdf, 2005. Accessed 13 March, 2007. 25

[60] Peter Dadam, Manfred Reichert, and K. Kuhn. Clinical workflows - the killer application for process-oriented information systems? In *Proceedings of the 4th International Conference on Business Information Systems*, pages 36–59, Poznan, Poland, April 2000. Springer-Verlag. 31, 32, 59

[61] Wolfgang Deiters, Thomas Goesmann, Katharina Just-Hahn, Thorsten Löffeler, and Roland Rolles. Support for exception handling through workflow management systems. In *Proceedings of the 1998 Conference on Computer-Supported Cooperative Work Workshop: Towards Adaptive Workflow Systems*, page 5, Seattle, Washington, USA, 1998. 35

[62] Barbara Dellen, Frank Maurer, and Gerhard Pews. Knowledge-based techniques to increase the flexibility of workflow management. *Data & Knowledge Engineering Journal*, 23(3):269–295, 1997. 1, 2, 28

[63] S.G. Deng, Z. Yu, Z.H. Wu, and L.C. Huang. Enhancement of workflow flexibility by composing activities at run-time. *Proceedings of the 2004 ACM symposium on Applied Computing*, pages 667–673, 2004. 33

[64] A. Dogac, Y. Tambag, A. Tumer, M. Ezbiderli, N. Tatbul, N. Hamali, C. Icdem, and C. Beeri. A workflow system through cooperating agents for control and document flow over the internet. In Opher Etzion and Peter Scheuermann, editors, *Proceedings of the 7th International Conference on Cooperative Information Systems (CoopIS 2000)*, volume 1901 of *Lecture Notes in Computer Science*, pages 138–143, Eilat, Israel, September 2000. Springer. 28

[65] 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, Australia, December 2000. 69

[66] M. Dumas, W.M.P. van der Aalst, and A.H.M. ter Hofstede, editors. *Process-Aware Information Systems: Bridging People and Software through Process Technology*. Wiley-Interscience, New York, 2005. 224, 236

[67] Johann Eder and W. Liebhart. The workflow activity model WAMO. In *Proceedings of the 3rd International Conference on Cooperative Information Systems (CoopIS '95)*, pages 87–98, Vienna, Austria, 1995. 34, 178

[68] Johann Eder, Euthimios Panagos, Heinz Pozewaunig, and Michael Rabinovich. Time management in workflow systems. In *Proceedings of the 3rd International Conference on Business Information Systems*, Poznan, Poland, April 1999. Springer. 24

[69] David Edmond and Arthur H.M. ter Hofstede. A reflective infrastructure for workflow adaptability. *Data & Knowledge Engineering*, 34:271–304, 2000. 57

[70] 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. 2

[71] Rainer Endl, Gerhard Knolmayer, and Marcel Pfahrer. Modeling processes and workflows by business rules. In *Proceedings of the 1st European Workshop on Workflow and Process Management (WPM'98)*, Zurich, Switzerland, October 1998. 38, 61

[72] Y. Engestrom. *Learning by Expanding: An Activity-Theoretical Approach to Developmental Research*. Orienta-Konsultit, Helsinki, 1987. 49, 50, 52, 53

[73] Gert Faustmann. Workflow management and causality trees. In *Proceedings of the Second International Conference on the Design of Cooperative Systems (COOP'96)*, pages 645–663, Juan-les-Pins, France, June 1996. NRIA Press, Rocqencourt. 39, 59

[74] B.R. Gaines. Induction and visualisation of rules with exceptions. In *Proceedings of the 6th Banff AAAi Knowledge Acqusition of Knowledge Based Systens Workshop*, pages 7.1–7.17, Banff, Canada, 1991. 71, 199

[75] Dimitrios Georgakopoulos, Mark Hornick, and Amit Sheth. An overview of workflow management: From process modelling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, 1995. 1

[76] Michael Georgeff and Jon Pyke. Dynamic process orchestration. White paper, Staffware PLC, March 2003. 25

[77] Paul Gibson. Formal object oriented requirements: simulation, validation and verification. In *Proceedings of the 13th European Simulation Multi-conference (ESM'99)*, pages 103–111, Warsaw, Poland, June 1999. SCS International. 92

[78] Edward Gould, Irina Verenikina, and Helen Hasan. Activity theory as a basis for the design of a web based system of enquiry for world war 1 data. 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, Sweden, 2000. 64

[79] D. Grigori, F. Charoy, and C. Godart. Enhancing the flexibility of workflow execution by activity anticipation. *International Journal of Business Process Integration and Management*, 1(3):143–155, 2006. 33

[80] C. Günther, S. Rinderle, M. Reichert, and W. van der Aalst. Change mining in adaptive process management systems. In R. Meersman and Z. Tari, editors, *Proceedings of the 14th International Conference on Cooperative Information Systems (CoopIS'06)*, volume 4275 of *Lecture Notes in Computer Science*, pages 309–326, Montpellier, France, November 2006. Springer-Verlag. 88

[81] Claus Hagen and Gustavo Alonso. Flexible exception handling in process support systems. Technical report No. 290, ETH Zurich, Switzerland, 1998. 6, 34, 62, 89

[82] Claus Hagen and Gustavo Alonso. Exception handling in workflow management systems. *IEEE Transactions on Software Engineering*, 26(10):943–958, October 2000. 23, 26

[83] Petra Heinl, Stefan Horn, Stefan Jablonski, Jens Neeb, Katrin Stein, and Michael Teschke. A comprehensive approach to flexibility in workflow management systems. In *Proceedings of International Joint Conference on Work Activities Coordination and Collaboration (WACC'99)*, pages 79–88, San Francisco, California, USA, February 1999. ACM. 24

[84] Horst Hendriks-Jansen. *Catching ourselves in the act : situated activity, interactive emergence, evolution, and human thought*. MIT Press, Cambridge, Mass, 1996. 45, 64

[85] 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 *Proceedings of the Conference on Extending Database Technology: Software Demonstration Track*, pages 29–30, Konstanz, Germany, March 2000. 26

[86] Hewlett-Packard Company, Palo Alto, CA. *HP Process Manager: Process Design Guide*, 4.2 edition, 2001. 26

[87] A.H.M. ter Hofstede and A.P. Barros. Specifying complex process control aspects in workflows for exception handling. In *Proceedings of the 6th International Conference on Database Systems for Advanced Applications*, pages 53–60, Hsinchu, Taiwan, 1999. IEEE Computer Society. 24

[88] Anatol W. Holt. *Organized Activity and Its Support by Computer*. Kluwer Academic Publishers, Dordrecht, 1997. 45

[89] Stefan Horn and Stefan Jablonski. An approach to dynamic instance adaptation in workflow management applications. In *Proceedings of the 1998 Conference on Computer-Supported Cooperative Work Workshop: Towards Adaptive Workflow Systems*, Seattle, Washington, USA, 1998. ACM. 35

[90] San-Yih Hwang, Sun-Fa Ho, and Jian Tang. Mining exception instances to facilitate workflow exception handling. In *Proceedings of the 6th International Conference on Database Systems for Advanced Applications*, pages 45–52, Hsinchu, Taiwan, April 1999. IEEE Computer Society. 5

[91] S.Y. Hwang and J. Tang. Consulting past exceptions to facilitate workflow exception handling. *Decision Support Systems*, 37(1):49–69, 2004. 88

[92] IBM WebSphere MQ Workflow: Concepts and architecture. http://publibfp.boulder.ibm.com/epubs/pdf/h1262857.pdf, 2005. Accessed 14 March, 2007. 25

[93] Future Strategies Inc. E-workflow - the workflow portal. http://www.e-workflow.org, April 2002. Accessed 17 September, 2005. 1

[94] David K. Irving and Peter W. Rea. *Producing and Directing the Short Film and Video*. Focal Press, Burlington, Oxford, United Kingdom, 3rd edition, 2006. 191

[95] K. Issroff and E. Scanlon. Case studies revisited: what can activity theory offer? In P. Dillenbourg, A. Eurelings, and K. Hakkarainen, editors, *European Perspectives on Computer-Supported Collaborative Learning. Proceedings of the First European Conference on Computer-Supported Collaborative Learning*, Maastricht, The Netherlands, 2001. 64

[96] Monique Jansen-Vullers and Mariska Netjes. Business process simulation. In *Proceedings of the Seventh Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*. University of Aarhus, Denmark, October 2006. http://www.daimi.au.dk/CPnets/workshop06/cpn/papers/Paper03.pdf. Accessed 30 April, 2007. 92

[97] Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, volume 1. Springer, Berlin, 2nd edition, 1997. 91, 92

[98] 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. 1

[99] Gregor Joeris and Otthein Herzog. Managing evolving workflow specifications. In *Proceedings of the 3rd IFCIS International Conference on Cooperative Information Systems (CoopIS '98)*, pages 310–319, New York, New York, USA, 1998. IEEE Computer Society. 67

[100] Stef Joosten. Conceptual theory for workflow management support systems. Technical report, Centre for Telematics and Information Technology, University of Twente, The Netherlands, 1995. 8, 10

[101] Stef Joosten and Sjaak Brinkkemper. Fundamental concepts for workflow automation in practice. In *Proceedings of the International Conference on Information Systems (ICIS'95)*, Amsterdam, The Netherlands, December 1995. 34

[102] Jens Bæk Jørgensen, Kristian Bisgaard Lassen, and Wil M. P. van der Aalst. From task descriptions via coloured petri nets towards an implementation of a new electronic pa-

tient record. In *Proceedings of the Seventh Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*. University of Aarhus, Denmark, October 2006. http://www.daimi.au.dk/CPnets/workshop06/cpn/papers/Paper03.pdf. Accessed 30 April, 2007. 92

[103] J. J. Kaasboll and O. Smordal. Human work as context for development of object-oriented modelling techniques. In S. Brinkkemper, editor, *IFIP WG 8.1/8.2 Working Conference on Principles of Method Construction and Tool Support*, pages 111–125. Chapman and Hall, Atlanta, Georgia, USA, 1996. 8, 30, 32, 59

[104] P.J. Kammer, G.A. Bolcer, R.N. Taylor, A.S. Hitomi, and M. Bergman. Techniques for Supporting Dynamic and Adaptive Workflow. *Computer Supported Cooperative Work (CSCW)*, 9(3):269–292, 2000. 5

[105] Byeong Ho Kang, Phil Preston, and Paul Compton. Simulated expert evaluation of multiple classification ripple down rules. In *Proceedings of the 11th Workshop on Knowledge Acquisition, Modeling and Management*, Banff, Alberta, Canada, April 1998. 68, 69, 72

[106] Victor Kaptelinin. *Computer-Mediated Activity: Functional Organs in Social and Developmental Contexts*, pages 45–67. In Nardi [131], 1996. 48, 64

[107] M. Khomyakov and I. Bider. Achieving workflow flexibility through taming the chaos. *Journal of Conceptual Modeling*, 21, August 2001. http://www.inconcept.com/JCM/August2001/bider.html. Accessed 23 October, 2004. 33

[108] Markus Kradolfer and Andreas Geppert. Dynamic workflow schema evolution based on workflow type versioning and workflow migration. In *Proceedings of the 1999 IFCIS International Conference on Cooperative Information Systems (CoopIS'99)*, pages 104–114, Edinburgh, Scotland, September 1999. IEEE Computer Society. 67

[109] Akhil Kumar and J.L. Zhao. A framework for dynamic routing and operational integrity controls in a workflow management system. In *Proceedings of the 29th Annual Hawaii International Conference on Systems Sciences*, pages 492–501, Hawaii, 1996. IEEE Computer Society. 32

[110] Kari Kuutti. *Activity Theory as a Potential Framework for Human-Computer Interaction Research*, pages 17–44. In Nardi [131], 1996. 46, 48, 49, 50, 51, 53, 54, 58, 64

[111] Bernie Laramie. The postproduction process. In Robert Benedetti, Michael Brown, Bernie Laramie, and Patrick Williams, editors, *Creative Postproduction: Editing, Sound, Visual Effects, and Music for Film and Video*, pages 7–64. Pearson/Allyn and Bacon, Boston, 2004. 193

[112] 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, Sweden, 2000. 4, 64

[113] John J. Lee and Rob Holt. *The Producer's Business Handbook*. Focal Press, Burlington, Oxford, United Kingdom, 2nd edition, 2006. 191

[114] Yu Lei and Munindar P. Singh. 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. 2, 5, 34

[115] A.N. Leontiev. The problem of activity in psychology. *Soviet Psychology*, 13(2):4–33, 1974. 46, 48

[116] A.N. Leontiev. *Activity, Consciousness and Personality*. Prentice Hall, Englewood Cliffs, NJ, 1978. 51

[117] Karl Leung and Jojo Chung. The Liaison workflow engine architecture. In *Proceedings of the 32nd Hawaii International Conference on System Sciences*, Maui, Hawaii, 1999. IEEE. 38

[118] Frank Leymann. Workflow-based coordination and cooperation in a service world. In R. Meersman and Z. Tari, editors, *Proceedings of the 14th International Conference on Cooperative Information Systems (CoopIS'06)*, volume 4275 of *Lecture Notes in Computer Science*, pages 2–16, Montpellier, France, November 2006. Springer-Verlag. 2

[119] L. Liu and C. Pu. ActivityFlow: Towards incremental specification and flexible coordination of workflow activities. In David W. Embley and Robert C. Goldstein, editors, *Proceedings of the 16th International Conference on Conceptual Modeling (ER'97)*, pages 169–182, Los Angeles, California, USA, November 1997. Springer. 27

[120] Zongwei Luo, Amit P. Sheth, Krys Kochut, and John A. Miller. Exception handling in workflow systems. *Applied Intelligence*, 13(2):125–147, 2000. 37

[121] Johannes Lux. Creating a reference model for the creative industries evaluation of configurable event driven process chains in practice. Master's thesis, Institute of Information Systems, Chair of Information Management, University of Münster, Germany, 2006. 193, 194

[122] Thomas W. Malone, Kevin Crowston, Jintae Lee, Brian Pentland, Chrysanthos Dellarocas, George Wyner, John Quimby, Charles S. Osborn, Abraham Bernstein, George Herman, Mark Klein, and Elissa O'Donnell. Tools for inventing organizations: Toward a handbook of organisational processes. *Management Science*, 45(3):425–443, March 1999. 32

[123] M. V. Manago and Y. Kodratoff. Noise and knowledge acquisition. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, volume 1, pages 348–354, Milano, Italy, 1987. Morgan Kaufmann. 68

[124] H. Mourão and P. Antunes. Exception handling through a workflow. In R. Meersman and Z. Tari, editors, *Proceedings of the 12th International Conference on Cooperative Information Systems (CoopIS'04)*, volume 3290 of *Lecture Notes in Computer Science*, pages 37–54, Agia Napa, Cyprus, 2004. Springer-Verlag. 36

[125] H. Mourão and P. Antunes. A Collaborative Framework for Unexpected Exception Handling. In Hugo Fuks, Stephan Lukosch, and Ana Carolina Salgado, editors, *Groupware: Design, Implementation and Use. Proceedings of the 11th International Workshop on Groupware (CRIWG 2005)*, volume 3706 of *Lecture Notes In Computer Science*, pages 168–183, Porto de Galinhas, Brazil, September 2005. Springer-Verlag. 36

[126] 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. 1, 23, 26, 62

[127] 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. 27

[128] Lewis Mumford. *Technics and Civilization*. Harcourt Brace Jovanovich, New York, 1963. 45

[129] Daisy Mwanza. Mind the gap: Activity theory and design. Technical report, KMi Technical Reports, KMI-TR95, http://kmi.open.ac.uk/-publications/techreports.html, Knowledge Media Institute, The Open University, Milton Keynes, United Kingdom, 2000. Accessed 23 February, 2000. 53, 64

[130] Bonnie A. Nardi. *Activity Theory and Human-Computer Interaction*, pages 7–16. In Nardi [131], 1996. 29, 46, 50, 58, 64

[131] Bonnie A. Nardi, editor. *Context and Consciousness: Activity Theory and Human-Computer Interaction*. MIT Press, Cambridge, Massachusetts, 1996. 233, 236

[132] Donald A. Norman. *The Invisible Computer*. MIT Press, Cambridge, Massachusetts, 1997. 46

[133] Hendrik Oberheid. A colored petri net model of cooperative arrival planning in air traffic control. In *Proceedings of the Seventh Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*. University of Aarhus, Denmark, October 2006. http://www.daimi.au.dk/CPnets/workshop06/cpn/papers/Paper03.pdf. Accessed 30 April, 2007. 92

[134] Andreas Oberweis. *Person-to-Application Processes: Workflow Management*, chapter 2, pages 21–36. In Dumas et al. [66], 2005. 2

[135] Pacific Knowledge Systems. Products: Rippledown, http://www.pks.com.au/products/validator.htm, September 2003. Accessed 23 April, 2002. 69

[136] Pallas Athena. Case handling with FLOWer: Beyond workflow. positioning paper. http://www.pallas-athena.com/downloads/eng_flower/flowerup.pdf, 2000. Accessed 13 May, 2003. 26, 40, 59

[137] Nathaniel Palmer. A survey of business process initiatives. http://wfmc.org/researchreports/Survey_BPI.pdf, January 2007. Accessed 4 April, 2007. 10, 11, 210

[138] M. Pesic and W.M.P. van der Aalst. A declarative approach for flexible business processes. In J. Eder and S. Dustdar, editors, *Proceedings of the First International Workshop on Dynamic Process Management (DPM 2006)*, volume 4103 of *Lecture Notes in Computer Science*, pages 169–180, Vienna, Austria, September 2006. Springer-Verlag, Berlin, Germany. 28

[139] Matthias Rauterberg. Activity and perception: an action theoretical approach. *Systematica*, 14(1):1–6, 1999. x, 29, 38, 39, 59, 64

[140] Matthias Rauterberg and Daniel Felix. Human errors: Disadvantages and adavantages. In *Proceedings of the 4th Pan Pacific Conference on Occupational Ergonomics*, pages 25–28, Hsinchu, Taiwan ROC, November 1996. Ergonomics Society Taiwan. 38, 64

[141] M. Reichert, P. Dadam, and T. Bauer. Dealing with forward and backward jumps in workflow management systems. *Software and Systems Modeling*, 2(1):37–58, 2003. 27

[142] M. Reichert, S. Rinderle, U. Kreher, and P. Dadam. Adaptive process management with ADEPT2. In *Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, pages 1113–1114, Tokyo, Japan, April 2005. IEEE Computer Society Press. 26

[143] Manfred Reichert and Peter Dadam. A framework for dynamic changes in workflow management systems. In *Proceedings of the 8th International Workshop on Database and Expert Systems Applications (DEXA 97)*, pages 42–48, Toulouse, France, 1997. IEEE Computer Society Press. 1, 5, 31, 32, 45, 59

[144] Manfred Reichert and Peter Dadam. ADEPT$_{flex}$ - supporting dynamic changes of workflows without losing control. *Journal of Intelligent Information Systems - Special Issue on Workflow Managament*, 10(2):93–129, 1998. 31, 59

[145] H.A. Reijers, J.H.M. Rigter, and W.M.P. van der Aalst. The case handling case. *International Journal of Cooperative Information Systems*, 12(3):365–391, 2003. 40, 41, 42, 59

[146] D. Richards, V. Chellen, and P. Compton. The reuse of ripple down rule knowledge bases: Using machine learning to remove repetition. In *Proceedings of the 2nd Pacific Knowledge Acquisition Workshop (PKAW'96)*, Coogee, Australia, October 1996. 199

[147] Debbie Richards. Combining cases and rules to provide contextualised knowledge based systems. In Varol Akman, Paolo Bouquet, Richmond H. Thomason, and Roger A. Young, editors, *Modeling and Using Context, Proceedings of the Third International and Interdisciplinary Conference, CONTEXT 2001*, volume 2116 of *Lecture Notes in Artifical Intelligence*, pages 465–469, Dundee, UK, July 2001. Springer-Verlag, Berlin. 68

[148] 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. 2

[149] U.V. Riss, A. Rickayzen, H. Maus, and W.M.P. van der Aalst. Challenges for Business Process and Task Management. *Journal of Universal Knowledge Management*, 0(2):77–100, 2005. http://www/jukm.org/jukm_0_2/riss. Accessed 27 January, 2007. 40, 59

[150] M. Rosemann and W.M.P. van der Aalst. A configurable reference modelling language. *Information Systems*, 32(1):1–23, 2007. 193

[151] A. Rozinat and W.M.P. van der Aalst. Conformance Testing: Measuring the Fit and Appropriateness of Event Logs and Process Models. In Christoph Bussler and Armin Haller, editors, *BPM 2005 Workshops (Workshop on Business Process Intelligence)*, volume 3812 of *Lecture Notes in Computer Science*, pages 163–176, Nancy, France, 2006. Springer Verlag. 29

[152] 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. 23, 24, 26, 77, 81

[153] Shazia Sadiq and Maria Orlowska. On capturing process requirements of workflow based business information systems. In *Proceedings of the 3rd International Conference on Business Information Systems (BIS99)*, Poznan, Poland, April 1999. 51, 58

[154] SAP advanced workflow techniques. https://www.sdn.sap.com/irj/servlet/prt/portal/prtroot/docs/library/uuid/82d03e23-0a01-0010-b482-dccfe1c877c4, 2006. Accessed 17 March, 2007. 25

[155] Thomas Schael. *Workflow management systems for process organisations*, volume 1096 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2nd edition, 1998. 46, 58

[156] Tobias Scheffer. Algebraic foundation and improved methods of induction of ripple down rules. In *Proceedings of the 2nd Pacific Rim Workshop on Knowledge Acquisition*, pages 279–292, Sydney, Australia, 1996. 69, 70, 71, 169, 199

[157] Ole Smordal. Classifying approaches to object oriented analysis of work with activity theory. In M. E. Orlowska and R. Zicari, editors, *Proceedings of the 4th International Conference on Object-Oriented Information Systems*, Brisbane, Australia, 1997. Springer-Verlag. 30, 52

[158] Lynn Andrea Stein. Challenging the computational metaphor: Implications for how we think. *Cybernetics and Systems*, 30(6), 1999. 44, 45, 64

[159] Remco van Stiphout, Theo Dirk Meijler, Ad Aerts, Dieter Hammer, and Riné le Comte. TREX: Workflow TRansactions by means of EXceptions. In J.Eder, H.J. Schek, and L. Salsa, editors, *Proceedings of the EDBT Workshop on Workflow Management Systems (WFMS'98)*, pages 21–26, Valencia, Spain, March 1998. 27

[160] Diane M. Strong and Steven M. Miller. Exceptions and exception handling in computerized information processes. *ACM Transactions on Information Systems*, 13(2):206–233, 1995. 4, 8, 58

[161] L. A. Suchman. *Plans and Situated Actions: The Problem of Human Machine Communication*. Cambridge, United Kingdom: Cambridge University Press, 1987. 29

[162] Gillian Symon, Karen Long, and Judi Ellis. The coordination of work activities: cooperation and conflict in a hospital context. *Computer Supported Cooperative Work*, 5(1):1–31, 1996. 58

[163] TIBCO iProcess Suite whitepaper. `http://www.staffware.com/resources/software/bpm/tibco_iprocess_suite_whitepaper.pdf`, 2006. Accessed 13 March, 2007. 25

[164] Dimitrios Tombros and Andreas Geppert. Building extensible workflow systems using an event-based infrastructure. In Benkt Wangler and Lars Bergman, editors, *Proceedings of the 12th International Conference on Advanced Information Systems Engineering (CAiSE 2000)*, volume 1789 of *Lecture Notes in Computer Science*, pages 325–339, Stockholm, Sweden, June 2000. Springer. 35

[165] Goce Trajcevski, Chitta Baral, and Jorge Lobo. Formalizing (and reasoning about) the specifications of workflows. In Opher Etzion and Peter Scheuermann, editors, *Proceedings of the 7th International Conference on Cooperative Information Systems (CoopIS 2000)*, volume 1901 of *Lecture Notes in Computer Science*, pages 1–17, Eilat, Israel, September 2000. Springer. 31

[166] Dennis Trewin. Televison, film and video production in Australia (publication 8679.0). Australian Bureau of Statistics http://www.ausstats.abs.gov.au/ausstats/subscriber.nsf/0/14F1A528655E8486CA256EDE00782780/File/86790_2002-03.pdf, 2004. Accessed 13 April, 2007. 191

[167] A.M Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936. 45

[168] Phil Turner, Susan Turner, and Julie Horton. Towards an activity-based approach to requirements definition. http://www.comp.lancs.ac.uk/computing/research/cseg/projects/coherence/workshop/Turner.html, April 1998. Accessed 7 February, 2004. 64

[169] Michael B. Twidale and Paul F. Marty. Coping with errors: the importance of process data in robust sociotechnical systems. In *Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work (CSCW'00)*, pages 269–278, Philadelphia, Pennsylvania, USA, December 2000. ACM Press. 9, 30

[170] L.S. Vygotsky. *Collected Works: Questions of the Theory and History of Psychology*, chapter Consciousness as a Problem in the Psychology of Behaviour. Moscow: Pedagogika, 1925/1982. 47

[171] B. Weber, W. Wild, and R. Breu. CBRFlow: Enabling adaptive workflow management through conversational case-based reasoning. In Peter Funk and Pedro A. González

Calero, editors, *Proceedings of the 7th European Conference for Adavances in Case Based Reasoning (ECCBR'04)*, volume 3155 of *Lecture Notes In Computer Science*, pages 434–448, Madrid, Spain, August 2004. Springer. 28

[172] J. Wertsch, editor. *The Concept of Activity in Soviet Psychology*. Armonk, NY: Sharpe, 1981. 47

[173] Workflow Management Coalition. Introduction to workflow. `http://www.wfmc.org/introduction_to_workflow.pdf`, 2002. Accessed 14 November 2004. 45

[174] Cong Yuan and Jonathan Billington. A colored petri net model of the dynamic MANET on-demand routing protocol. In *Proceedings of the Seventh Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*. University of Aarhus, Denmark, October 2006. `http://www.daimi.au.dk/CPnets/workshop06/cpn/papers/Paper03.pdf`. Accessed 30 April, 2007. 92