ON CORRECTNESS CRITERIA FOR WORKFLOW EXCEPTION HANDLING POLICIES

Belinda M. Carter and Maria E. Orlowska
School of Information Technology and Electrical Engineering, University of Queensland, Australia

Keywords: Workflow, exception handling, verification, correctness criteria, event-driven business process management.

Abstract: Exception handling during the execution of workflow processes is a frequently addressed topic in the literature. Exception handling policies describe the desired response to exception events with respect to the current state of the process instance in execution. In this paper, we present insights into the definition and verification of such policies for handling asynchronous, expected exceptions. In particular, we demonstrate that the definition of exception handling policies is not a trivial exercise in the context of complex processes, and, while different approaches to defining and enforcing exception handling policies have been proposed, the issue of verification of the policies has not yet been addressed. The main contribution of this paper is a set of correctness criteria which we envisage could form the foundation of a complete verification solution for exception handling policies.

1 INTRODUCTION

Workflow technology is ideal for supporting highly repetitive and predictable processes. However, many processes are faced with the need to deal with exceptional situations that may arise during their execution (Casati 1999). Workflows may be affected by different types of exceptions: *system* failures such as hardware and software crashes and *logical* failures or exceptions. Logical failures refer to application-specific exceptional events for which the control and data flow of a workflow is no longer adequate for the process instance (Müller, et al 2004). Many logical failures may be *unexpected*, and these must be handled manually on an ad hoc basis by knowledge workers. However, many exceptions are *expected* – the inconsistencies between the business process in the real world and its corresponding workflow representation can be anticipated, even if they might not be frequent (Casati and Pozzi 1999). That is, workflows describe the 'normal behaviour' (forming the 'core') of a process whereas expected exceptions model the 'occasional behaviour'.

Expected exceptions can be *synchronous* with respect to the flow of work, but most often they are *asynchronous* – that is, they can be raised at an arbitrary stage of the process, potentially during a long-duration activity (Casati and Pozzi 1999). Cancellations of customer orders and car accidents during a rental process are examples of asynchronous events.

In some applications, there may be one standard desired response to the occurrence of such an exception event, regardless of the execution state of the underlying process instance. However, in most real world scenarios, the required reaction to these events depends on the data associated with the relevant process instance (that is, its execution state).

As discussed in the next section, a myriad of exception handling approaches exist, each with its own expressivity, syntax and visual representation. However, regardless of the approach that is adopted, exception handling, in essence, still constitutes the enforcement of a set of exception handling rules defined in terms of the state of the process instance.

Exceptional situations are usually very complicated (Luo, et al 2000) and we argue that it is very easy to define policies that may produce unintended execution behaviour. However, while most authors are fast to claim that exception handling is complex, and it is well known that the analysis of rules in general is non-trivial, little attention has been paid to the issue of verification of these rules. This issue is glossed over in the literature, so we attempt to address it in this paper.

The focus of this paper is on the definition and verification of policies for handling expected

M. Carter B. and E. Orlowska M. (2007).
ON CORRECTNESS CRITERIA FOR WORKFLOW EXCEPTION HANDLING POLICIES.
In *Proceedings of the Ninth International Conference on Enterprise Information Systems - DISI*, pages 315-322
DOI: 10.5220/0002374303150322
Copyright © SciTePress

exceptions that are based on external events that occur asynchronously with respect to the process. The policies make reference to the 'state' of the process instance, and, while there are multiple dimensions to this notion, we argue that two of these are common to many and indeed completely satisfy a large proportion of exception handling cases – the *position* of the process instance with respect to the process model (i.e. the currently executing activities), and the values of associated *case data*. And regardless of the particular exception handling approach that is adopted, this business logic must be captured and then processed at runtime.

Since this logic relates directly to the business itself, validation of the exception handling policies is largely a semantic issue to be performed by a domain expert and thus cannot be automated. However, we argue that there are some common principles of a generic nature that should be observed in order to perform the first step towards verification of the rules in an automated manner.

In the following sections, we summarize the related work, then present an introduction to the basic principles of workflow specification and execution. We introduce the notion of an exception handling policy and illustrate the concept with a simple but meaningful example. We then describe a set of generic properties that should be satisfied in order to ensure the 'correctness' of the policy. We conclude with an outlook for future research.

2 RELATED WORK

Exception handling is not a new concept, and has attracted considerable attention in the literature. Many approaches for flexible process enforcement have been proposed. The first approach is to encode the entire workflow process as a set of rules, thereby ensuring complete flexibility. For example (Bae, et al 2004) and (Kappel, et al 1997) present approaches where the process is described through a set of *Event-Condition-Action* (ECA) rules (c.f. (Widom and Ceri 1996)). However, while processes encoded through rules enable all predefined behavior to be enforced, it is well known that large sets of rules can interact in unknown ways (e.g. (Widom and Ceri 1996)). The importance of ensuring correctness of the process model before deployment has been emphasized in (Sadiq and Orlowska 2000b).

As already noted, this paper primarily addresses the issue of policy definition for handling expected exceptions that are based on external events that occur asynchronously with respect to the process. (Readers are referred to (Mourão and Antunes 2004) for a framework to support ad hoc interventions when dealing with unexpected exceptions.) Essentially, there are two approaches for incorporating exceptional cases into a process model – 'exception rules' and 'exception workflows' (Sadiq and Orlowska 2000a). The first approach is to implement exceptions through an explicit exception rule base. Each exception is modelled by an ECA rule, where the event describes the occurrence of a potentially exceptional situation, the condition verifies that the occurred event actually corresponds to an exception that must be managed, and the action reacts to the exception (Casati 1999). A different approach is presented in (Müller, et al 2004) where the core process is dynamically modified at run-time based on a set of rules – when exceptional events occur during process execution, the AgentWork system identifies the workflow instances to be adapted, determines the change operations to be applied, and automatically performs the change for those instances.

Alternatively, exceptions can be modelled as workflow processes themselves (Sadiq and Orlowska 2000a). This approach is taken in (Adams, et al 2005), which introduces the notion of Worklets, which are 'an extensible repertoire of self-contained sub-processes and associated selection and exception handling rules'. Choosing the most applicable worklet to be executed in response to an exception is achieved by evaluating conditions that are associated with each worklet. These conditions are defined using a combination of current data attribute values and the current state of each of the worklets that comprise the process instance. It was noted that the set of states for a worklet-enabled process may be deduced by mining the process log file (c.f. Aalst, et al 2003) but that full exploration of the specification of such conditions is yet to be completed.

Another approach is to consider the exception handling processes as sub-processes within the core process. In the 'event node approach' discussed in (Casati 1999), the workflow model includes a particular type of node, called an event node, which is able to observe asynchronous events and to activate its successor in the workflow graph when the event is detected. However, once again, it was noted that upon observation of an event, conditions 'can be used to select, among several exception management alternatives, the most adequate to deal with the current workflow state' (Casati, et al 1999). The complexity of defining such conditions was briefly emphasised in (Carter and Orlowska 2007) but the issue of definition and verification of exception handling policies has not yet received adequate attention in the literature.

3 BACKGROUND OF WORKFLOW SPECIFICATION AND EXECUTION

Before we discuss exception handling, let us first briefly summarize the basic principles of workflow specification and execution that are required for the subsequent discussion. A *workflow management system* (***WFMS***) is a system that completely defines, manages, and executes workflows. Before a workflow process can be enacted, it must be specified to reflect the process requirements. The process model describes the order of execution of tasks according to the business policies and resource/temporal constraints. Each task (activity) is a logical unit of work within a process that may be either manual or automated but performed by a single workflow participant.

The workflow model (W) is defined through a directed acyclic graph (DAG) consisting of nodes (N) and flows (F). Flows represent the control flow of the workflow. Thus, $W = <N, F>$ is a directed graph where N is a finite set of nodes, and F, $F \subseteq N \times N$, is a flow relation. Nodes are classified into tasks (T) and coordinators (O), where $O \cup T = N$ and $O \cap T = \varnothing$. Furthermore, we assume that the model is structurally correct (free from deadlocks and potential lack of synchronisation) according to the correctness criteria proposed in (Sadiq and Orlowska 2000b) for the purposes of this paper.

In this paper, we will adopt graphical process modelling notation whereby rectangles represent tasks, and forks and synchronizers (concurrent branching constructs), and ovals represent choices and merges (alternative branching constructs).

Let a workflow graph $W = <N, F>$ be given.

A ***Process Instance (PI)*** is a particular occurrence of the process. Let $I^W = \{i^W_1, i^W_2, \ldots, i^W_g\}$ represent the set of all instances of the process W. Note that for a commercial WFMS, g is likely to be in the order of hundreds of thousands.

For readability and simplicity of presentation, we consider one process so omit W as an index for all terms defined below (without loss of generality).

Each process instance is associated with three types of data. During process execution, the WFMS maintains internal ***Control Data*** that includes the internal state information associated with the execution of activities in the process. There are also two types of data that flow between activities. *Workflow Application Data* is manipulated directly by the invoked applications. *Workflow Relevant Data* (also known as '***Case Data***'), is the only type of application data accessible to the WFMS, and can be thought of as a set of global variables.

Data Items. Let D be the set of data variables $\{d_1, d_2, \ldots, d_q\}$ that are required as input to one or more nodes in T in order to execute instances of the process described by W. We refer to d_k, $k = 1 \ldots q$, as the *Data Items* for W.

Data Values. Corresponding to each data item $d_k \in D$ is a set V_k , $k = 1 \ldots q$ called the *Data Values* for d_k. and denoted $values(d_k)$. The data values are arbitrary, nonempty finite or countably infinite sets. Let $V = V_1 \cup V_2 \cup \ldots \cup V_n$.

The value for each data item $d \in D$ can be given at process instantiation or generated during process execution. Let $D_n \subseteq D$ represent the set of data items for which values are produced by node $n \in N$. (*Aside*: In practice, all nodes that produce data values will be activities.)

Time. A discrete time base is assumed such that $t \in \Im$ denotes a time point where $\Im$ is isomorphic with the set of natural numbers.

Case Data. Let $Z_i(t) \subset D \times V$ be the workflow case data for PI, $i \in I$ at time t such that $\forall (d_k, v_j) \in Z_i(t)$, $v_j \in (values(d_k) \cup NULL)$, and $\forall (d_k, v_j), (d_l, v_m) \in Z_i(t), d_k = d_l \rightarrow v_j = v_m$. The data values can be given at process instantiation or generated during process execution. The *NULL* value indicates that the data value has not (yet) been generated for a data item for i at time t.

Data Condition. Let C be the set of first order predicate expressions defined on the case data. That is, $\forall c \in C, i \in I, t \in \Im, c(Z_i(t)) \in \{True, False\}$. We refer to each $c \in C$ as a *Data Condition*. Data conditions may reference one or more data items/values.

Generally, there will be minor variations in the way in which different PIs are to be enacted. Specifically, not all of the activities in the process model will necessarily be relevant for every PI. Choice coordinators allow these variations to be represented in the same process model – a data condition $c \in C$ is associated with the outgoing flow of each choice node, and the associated flow is taken for a process instance $i \in I$ if and only if $c(Z_i(t)) = True$ at the time t at which i reaches the choice construct during its execution.

Process Instance Type (PIT). We call a class of PIs that execute the same set of tasks during their complete execution a *Process Instance Type (PIT)*. Let $\{I_1, I_2, \ldots, I_r\}$ be a partition of I that represents the set of PITs associated with W. Corresponding to each PIT I_k, $\forall k = 1 \ldots r$, is a set of data conditions $C_k \subset C$ that are associated with the choice paths taken by all instances $i \in I_k$. This set of data conditions uniquely identifies each PIT.

Since case data may be either given at process instantiation or generated during process execution, it is possible that at time t during the execution of an instance $i \in I$, the PIT of i is not yet determined. However, it must have been determined prior to completion of the process. Also, at time t during the execution of a WFMS (and indeed, during the lifetime of the system), it is possible that no PIs corresponding with one or more PITs have (yet) been facilitated, because one or more choice paths have not (yet) been taken – this depends entirely on the case data associated with the PIs that have been enacted by the system.

Process Instance Graph (PIG). Corresponding to each PIT I_k is a workflow graph $W_k = <N_k, F_k>$ where $N_k \subseteq N$, $F_k \subseteq N_k \times N_k$ called the *Process Instance Graph (PIG)* for I_k. This graph W_k is a structurally correct workflow graph that represents the exact process that is performed by all instances of the PIT I_k. Therefore, choice coordinators are not present in any PIG (and since the models are assumed to be structurally correct according to the criteria presented in (Sadiq and Orlowska 2000b), merge coordinators are also excluded from PIGs). The set of PIGs is known at 'design time', prior to the introduction of any process instances into the WFMS. The original process model W can be reconstructed from all PIGs $\{W_1, W_2, \dots W_r\}$ and the data conditions $\{C_1, C_2, \dots C_r\}$ associated with each of the corresponding instance types $\{I_1, I_2, \dots I_r\}$. There are two PIGs in the workflow graph depicted in Figure 1. The tasks associated with the PIG identified through condition c_1 are shaded.

Due to parallel branching structures, multiple tasks may be executing concurrently for any PI. Information about currently executing tasks is maintained as part of the Control Data.

Process Instance Position (PIP). Let $\mathcal{P} = \{T_1, T_2, \dots, T_s\}$ where $\forall\ b = 1\dots s$, $T_b \subset T$, be the set of sets of tasks that may be executing concurrently at any point in time if the correct semantics of all workflow modeling constructs in W are observed. We call $\mathcal{P}$ the set of *Process Instance Positions (PIPs)* for W. Note that since each PI $i \in I$ executes a single PIG W_k, it follows that $T_b \subseteq N_k, \forall\ b = 1\dots s$. Furthermore, the flow relation F_k enforces ordering constraints between all nodes $n \in N_k$, so $\forall\ n_1, n_2 \in T_b$, n_1 is not reachable from n_2 through F_k, and vice versa.

Clearly, not every combination of tasks is a PIP. For example, for the process model depicted in Figure 1, the PIPs are: *(1), (2, 4), (3, 4), (2, 5, 6), (2, 5, 7), (2, 8), (3, 5, 6), (3, 5, 7), (3, 8), (2, 9), (2, 10), (3, 9), (3, 10)* and *(11).* Note that the set of tasks *(2, 4, 9)* is not a PIP since these tasks will never be active at the same time due to the choice construct in the process model.

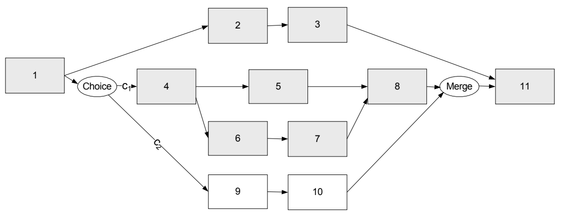

Figure 1: Example Process Model for Discussion.

e say that a node $n \in N$ is *prior to* ('<') a PIP $P \in \mathcal{P}$ if n is *prior to* any node $m \in P$, that is, $(n, m) \in F^*$ where F^* is the transitive closure of F.

Let T be a function: $I \times \Im \rightarrow \mathcal{P}$ that returns the set of currently executing tasks $P \in \mathcal{P}$ for process instance $i \in I$ at time $t \in \Im$.

Process Instance State ('State'). We will refer to the combination of PIP and case data for a PI $i \in I$ at time t as its *state*.

With this background, we now introduce the concept of exception handling policies.

4 EXCEPTION HANDLING POLICIES

Exception handling behavior depends on the underlying business requirements. Due to the costs associated with manual exception handling, this behaviour should be automated whenever possible. Before the system can automatically react to exception events, the desired exception handling behaviour must first be captured and encoded in a format that the system can interpret (similar to a process model for the core process). Each exception event has an associated 'Exception Handling Policy' that describes the desired reaction if the exception event is observed while the PI is in various execution states. In this section, we formally introduce the notion of an exception handling policy.

Assume the workflow graph W as introduced in the previous section is given. All introduced concepts correspond to W, unless otherwise noted.

Events. Let E be the set of all *Events* related to W for which exception handling policies are to be defined.

Actions. Let A be the set of all *Actions* to be performed in order to 'handle' each of the events $e \in E$ if and when they occur, as dictated by the business requirements. Each of these actions can be rep-resented through a process model, but definition of these models is outside the scope of this paper.

For a given event $e \in E$, an Exception Handling policy associates an action $a \in A$ with each PI state (that is, combination of PIP and the case data).

Exception Handling Policy. Let $\Delta^e : \mathbb{S}^e \rightarrow A$ be a function where $\mathbb{S}^e \subseteq \mathcal{P} \times C$. We call Δ^e the *Exception Handling Policy ('Policy')* for each event $e \in E$. Each *Policy Rule ('PR')* for each $e \in E$ identifies an action $a \in A$ to be performed $(P, c) \rightarrow a$ iff e is observed at time t during the execution of PI i when $\mathrm{T}(i, t) = P$ and $c(Z_i(t)) = True$.

Note that $\forall (P, c) \in \mathbb{S}^e$, (P, c) represents a set of process instance states. We say that a data condition $c \in C$ is '*associated with*' a PIP $P \in \mathcal{P}$ iff $(P, c) \in \mathbb{S}^e$.

When an exception event e is observed during the execution of a process instance i at time t, the system must retrieve the relevant exception handling policy Δ^e. Then, a query must be executed on the control data to determine the current PIP of i, that is, $P = \mathrm{T}(i, t)$, and then to retrieve the set of PRs for P from the policy. If there is one PR for P, the associated action is to be performed. If there are multiple PRs for P, each data condition associated with P is to be evaluated against the case data for i until a satisfied condition is found, at which point the action associated with the satisfied condition is to be performed.

5 EXAMPLE

We now illustrate the notion of an event handling policy with a simple but meaningful example. Consider the 'just in time' ordering process for computer systems, depicted in Figure 2. (Note that tasks are numbered for reference purposes only.)

Consider that $e \in E$ represents a cancellation of the order. The process has one data item: *Type*, such that *values(Type)* = {*'A'*, *'C'*}, that represent an account purchase and a credit card purchase, respectively. The value for *Type* is given at process instantiation (i.e. it comes with the order). Two data conditions are relevant: c_1: *Type = 'A'*, c_2: *Type = 'C'*. (We assume here that the account and credit card debits are approved for the sake of illustration.)

The business would like to enforce that the process can only be cancelled if the assembly of the system has not yet started, because they are unable to return goods once the packaging has been opened. If cancelled before that point, the core process should be halted, the hardware and software procurement semantically compensated for (if applicable), and the payment refunded – that is, either an account credit or a credit card refund issued. If received after this point, the cancellation is to be rejected. The policy for e could thus be defined as presented in Table 1. Note that the set of actions includes the following: *A1*: Issue Account Credit, *A2*: Issue Credit Card Refund, and *A3*: Reject Cancellation. Each row of the table represents a PR.

Table 1: Example Policy for Order Cancellation.

PIP	Data Conditions	Action
2	Type = 'A'	Cancel 2
3	Type = 'A'	Cancel/Compensate 3
4	Type = 'C'	Cancel/Compensate 4
5	Type = 'A'	Cancel 5 and Perform A1
5	Type = 'C'	Cancel 5 and Perform A2
6, 7	Type = 'A'	Cancel/Compensate 6, Cancel/Compensate 7, and Perform A1
6, 7	Type = 'C'	Cancel/Compensate 6, Cancel/Compensate 7, and Perform A2
6, 8	Type = 'A' or 'C'	Peform A3
9	Type = 'A' or 'C'	Peform A3
10	Type = 'A' or 'C'	Peform A3

Note that the control data must be inspected at a finer level to determine how much compensation is required (if any) for tasks, depending on the progress that has been made through their execution at the time the exception occurs. Now let us consider the verification of the defined policies.

6 CORRECTNESS OF EXCEPTION HANDLING POLICIES

Verification of process models prior to deployment is essential in order to detect potentially costly erroneous execution situations before they arise (Sadiq and Orlowska 2000b). Due to the

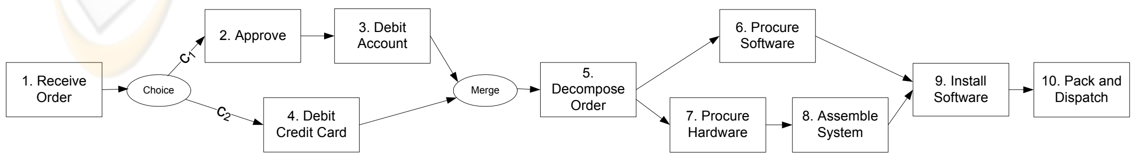

Figure 2: Ordering Process Example.

combinatorial explosion of PITs, it is inherently difficult to reason about the correctness of complex processes.

Just as it is possible to define processes that are erroneous according to specific correctness criteria, it is possible to define policies that can be determined to be erroneous prior to system deployment. And although exception handling is a peripheral issue to the enactment of the core process, it is equally important that exception handling policies be verified. Due to the infrequency of exception handling behaviour, it is perhaps even more important to verify the policies, because many months or even years may pass before particular situations arise for the errors to be detected. In this section, we introduce and describe the correctness criteria for exception handling policies.

Since the exception handling logic relates directly to the business itself, validation of the exception handling policies is largely a semantic issue to be performed by a domain expert and thus cannot be automated. However, we argue that there are two essential and two desirable properties of 'correct' exception handling policies. These generic properties for the correctness of a policy definition can be automatically verified as a precursor to semantic validation.

The essential requirements are

- *Deterministic Behaviour*, and
- *Data Availability*.

The desirable properties are

- *Completeness*, and
- *No Redundant Rules*.

While the expressiveness of the languages used to describe the policies and the particular modeled application domains may differ, we argue that these requirements should be observed. To our knowledge, no previous work has focused on the verification of the policies in this way. We will now introduce each of the correctness criterion in turn.

6.1 Deterministic Behaviour

The first correctness criterion for exception handling policies is that they must produce deterministic exception handling behaviour such that if an exception event is observed, the behaviour of the system can be determined based on the PIP and case data of the associated PI only.

Practically, this means that in a policy for a given event, the set of data conditions associated with each PIP (that is, across all PRs) must be mutually exclusive. That is, $\forall e \in E, P \in \mathcal{P}, c_1 \in C, c_2 \in C$ such that $(P, c_1) \in \mathbb{S}^e$ and $(P, c_2) \in \mathbb{S}^e$, if $\exists\, i \in I,\ t \in \Im$ such that $\mathrm{T}(i, t) = P$ and $c_1(Z_i(t)) = True$ and $c_2(Z_i(t)) = True$ then $c_1 = c_2$.

For example, if <*2, Type = 'A', Cancel Approval Request*> is a PR then <*2, Type = 'A', Perform A3*> is not a valid PR. If multiple actions are to be performed then they should be combined in one PR.

The effect on system behaviour resulting from a violation of this property depends on the specific implementation. In the best case scenario, the entire set of 'triggered' actions will be performed each time. Another possibility is that only one action is performed, but it is the same action each time. Whether this behaviour is correct is debateable, and the policy definition is not an accurate source of process knowledge in any case, but consistent behaviour would be observed. In the worst case, one action could be arbitrarily selected each time, which would result in inconsistent system behaviour that may remain undetected indefinitely. Another possibility is complete process or system suspension.

6.2 Data Availability

Data availability is a requirement for correct process execution. The problems associated with 'missing data' were discussed in (Sadiq et al 2004) in the context of execution of the 'core' business process. Specifically, it was noted that a data item (value) must be produced in a node of the graph prior to each node (that is, an activity or a choice node) that requires that data item (value) in order to execute.

This natural requirement extends to exception handling. During execution, data conditions must be evaluated only to resolve conflicts in order to determine which exception handling action to perform if multiple actions are defined for one PIP. Therefore, for any such PIPs, the data that is required in order to evaluate all associated data conditions need not be given at process instantiation, but must be generated during the execution of the process by the time the PI reaches that PIP.

That is, data availability requires that $\forall e \in E, \forall (P, c) \in \mathbb{S}^e$ where $P \in \mathcal{P}, c \in C$, the values for all data items referenced in c must be generated in a node that is prior to P. Therefore, the following must hold if all required data is available during exception handling: $\forall e \in E,\ \forall (P, c) \in \mathbb{S}^e$ where $P \in \mathcal{P}, c \in C,\ \forall\, d \in D$ such that d is referenced in c, $\exists\, n \in N$ such that $d \in D_n$ and $\exists\, m \in P$ such that $n < m$.

In our example, the only data item referenced in the policy rules is *Type* and its value is given at process instantiation, so all data is available.

The consequences of unavailable data depend on the implementation. If the values are '*NULL*' for all items on instantiation (as implied in this paper), then unanticipated condition evaluation results may result in incorrect exception handling. However, if the actual data items themselves are created during execution, system suspension is a likely outcome.

6.3 Completeness

On occurrence of a particular event $e \in E$ during the execution of a PI, Δ^e is consulted to determine the exception handling action to be performed for the PI, according to its current state. However, there may be some application scenarios whereby there are no actions to be performed on observation of an event for particular states of the PI in execution. It may be natural to think that no PRs should be defined for these states.

However, we argue that a thorough consideration of all possible states for a PI during the definition of exception handling policies is a complex task. It is therefore plausible and perhaps even likely that process designers may inadvertently omit one or more states when defining policies. A case could thus be made for requiring the definition of a PR for all such states with a corresponding action of 'No Action', in order to make it explicit that all states were indeed considered during the definition of the policy but that exception handling behaviour is not required in particular situations. This would help to ensure that the exception handling policies are an accurate and complete source of 'process knowledge', making it easier to understand and ultimately maintain the policies.

We therefore propose that such practices be recommended. However, while implementation-specific issues are outside the scope of this paper, it should be noted that it is possible to design the system such that an 'incomplete' policy neither suspends nor produces abnormal process or system execution, and so it is only a desirable, not critical, requirement that the policies are complete.

In order to ensure that the policies are complete, the following correctness criterion must hold: $\forall e \in E$, $i \in I$, $t \in \Im$, $\exists$ $(P, c) \in \mathbb{S}^e$ such that $c \in C$, $P \in \mathcal{P}$, such that $\mathrm{T}\,(i, t) = P$ and $c\,(Z_i(t)) = True$.

In our example, an action is required to be performed on observation of the cancellation event, regardless of the PI state for the order. The example policy, as presented in Table 1, is complete.

6.4 No Redundant Policy Rules

Ultimately, the main reason for the definition of exception handling policies is to facilitate their automatic enforcement. The 'complexity' of the definition is therefore not a critical issue, provided that the desired behaviour is achieved. However, exception handling policies also serve as a valuable source of 'process knowledge' that must be agreed upon and ultimately maintained by domain experts. As such, the policies should be as clear as possible. Being explicit with all assumptions (for example, with a 'complete' policy, as described above) partially addresses this issue.

Another desirable quality for policies is that they are not unnecessarily complex. Clearly, this issue is subjective to some degree, and is dependent on both the syntax and semantics of the PR condition language (which is not the focus of this paper) and also the underlying business scenario to be facilitated (which can only be considered by domain experts and process modellers). However, it is possible to make a generic observation in that policies in which one or more PRs are completely redundant (that is, they will *never* be executed, regardless of the state of the relevant PI) should be avoided. Redundant PRs can be safely removed without any impact on exception handling behaviour.

Not only do redundant rules make the policy more difficult to comprehend, but they may also negatively affect system performance, since more PRs than necessary must be searched through to find the relevant action. However, this effect could be minimised with appropriate indexing, and exception handling behaviour should not occur frequently anyway (by definition). In any case, as for completeness, the presence of redundant PRs will generally not cause undesirable exception handling behaviour and so this is not a critical requirement for policies.

The set of the PIPs $\mathcal{P}$ is 'valid' in the sense that it is constructed after analysis of the underlying process model – that is, they are not random combinations of nodes. However, the PRs will be redundant if the associated data condition is a contradiction. Detecting this issue is a relatively easy task, by inspecting the data condition in isolation, knowing the domain values for each of the referenced data items. However, we wish to make a more subtle observation.

In particular, we observe that a PR is redundant if the *combination* of PIP and data condition in the PR is invalid. Such a situation is possible because the concepts of PIP and case data (on which data conditions are defined) are not orthogonal. Consider a process instance state $(P, c) \in \mathbb{S}^e$ such that $e \in E$, $P \in \mathcal{P}$, $c \in C$ for which a PR is defined. Recall that every PIP is applicable for a subset of PIGs, each of which is associated with a set of data conditions that are satisfied by every PI that executes the PIG. Therefore, each PIP $P \in \mathcal{P}$ is associated with a set of conditions $C^P \subset C$ that are satisfied by every PI that reaches P. We can encapsulate C^P into a single logical expression (data condition) $h = c_1 \wedge c_2 \wedge \ldots \wedge c_y$, $\forall\, c_{1..y} \in C^P$. We note if the expression $h \wedge d$ is a contradiction then the policy rule is redundant.

For example, the policy rule <*4, Type = 'A', Perform A3*> is redundant, since all PIs that reach PIP *4* must satisfy the condition *Type = 'C'*.

7 CONCLUSIONS AND FUTURE WORK

Workflow technology is ideal for supporting repetitive and predictable processes, but exceptions occur often during the execution of processes in the real world. An important class of such exceptions is those that are *expected* and are *asynchronous* with respect to the process in execution. The desired reaction in response to these events will often depend on the current state of process execution, and we argue that the important aspects of this state for the majority of exception handling situations are the position of the process instance through the process model and the values of the case data at the time at which the exception event occurs. This reactive behaviour is encapsulated in an exception handling policy for each event.

In this paper, we have demonstrated that the definition of such policies for complex processes is a challenging exercise. In particular, it is possible to introduce errors into the policies that may result in undesirable execution behaviour if they were to remain undetected. It is therefore essential to verify policies prior to deployment, and this issue has not yet been addressed in the literature.

We have argued that regardless of the expressiveness or ‘syntactic sugar’ of the language used to define the policies, or the modelled application domain, a set of generic requirements must be satisfied. The presentation of correctness criteria for exception handling policies is a major contribution of this paper. We envisage that these properties will form the foundation for a complete verification solution for policies, to be utilised before the policies are semantically validated by the domain experts.

In our future work, we will develop a methodology for the automated verification of exception handling policies based on the proposed correctness criteria. We will also relax the restriction that state is described only through position and case data and consider other types of workflow control data in policy definition and subsequent verification. Finally, we will consider the development of a software tool to assist with the specification and verification of policies.

REFERENCES

van der Aalst, W.M.P., van Dongen, B.F., Herbst, J., Maruster, L., Schimm, G. and Weijters, A.J.M.M. 2003, ‘Workflow mining: A survey of issues and approaches’, *Data and Knowledge Engineering*, vol. 47, no. 2, pp. 237-267.

Adams, M., ter Hofstede, A. H. M., Edmond, D. and van der Aalst, W.M.P. 2005, ‘Facilitating Flexibility and Dynamic Exception Handling in Workflows through Worklets’, in Proc. *17th Conference on Advanced Information Systems Engineering (CAiSE05) Forum*, June 2005, Porto, Portugal.

Bae, J., Bae, H., Kang, S.-H. Kim, Y. 2004, ‘Automatic Control of Workflow Processes Using ECA Rules’, *IEEE Transactions on Knowledge and Data Engineering,* vol. 16, no. 8.

Carter, B. M. and Orlowska, M. E. 2007, ‘On the Definition of Exception Handling Policies for Asynchronous Events in Workflow Processes’, *IRMA International Conference*, Vancouver, British Columbia, Canada.

Casati, F. 1999, ‘A discussion on approaches to handling exceptions in workflows’, *SIGGROUP Bull*., vol. 20, no. 3, pp. 3-4.

Casati, F., Ceri, S., Paraboschi, S., and Pozzi, G. 1999, ‘Specification and implementation of exceptions in workflow management systems’, *ACM Transactions on Database Systems (TODS)* 24, pp. 405–451.

Casati, F., Pozzi, G. 1999, ‘Modeling Exception Behaviors in Commercial Workflow Management Systems’, in Proc. *4th International Conference on Cooperative Information Systems*, Edinburgh, Scotland.

Kappel, G., Rausch-Schott, S., and Retschitzegger, W. 1997, ‘Coordination in Workflow Management Systems - A Rule-Based Approach’, *Coordination Technology for Collaborative Applications - Organizations, Processes, and Agents,* LNCS, vol. 1364. Springer-Verlag, London, pp. 99-120.

Luo, Z., Sheth, A., Kochut, K., and Miller, J. 2000, ‘Exception handling in workflow systems’, *Applied Intelligence*, vol. 13, no. 2, pp. 125-147.

Mourão, H., Antunes, P. 2004, ‘Exception Handling Through a Workflow’, in Proc. *12th International Conference on Cooperative Information Systems* (CoopIS’04), pp. 37-54.

Müller, R., Greiner, U., & Rahm, E. 2004, ‘AgentWork: A Workflow-System Supporting Rule-Based Workflow Adaptation’, *Data and Knowledge Engineering*, vol. 51, no. 2.

Sadiq, S. and Orlowska, M.E. 2000a, ‘On Capturing Exceptions in Workflow Process Models’, in *Proc. 4th International Conference on Business Information Systems,* Poznan, Poland.

Sadiq, S., Orlowska, M., Sadiq, W., and Foulger, C. 2004, ‘Data Flow and Validation in Workflow Modeling’, in *Proc. Fifteenth Australasian Database Conference*, Dunedin, New Zealand.

Sadiq, W. and Orlowska, M.E. (2000b) Analyzing Process Models using Graph Reduction Techniques. *Information Systems*, vol. 25, no. 2, pp. 117-134. Elsevier Science.

Widom, J. and Ceri, S. 1996, *Active Database Systems,* Morgan Kaufmann Publishers.

Workflow Management Coalition. 1998, *The Workflow Reference Model*, Document Number TC00-1003, Issue 1.1, 19-Jan-95.