On the Definition of Exception Handling Policies for Asynchronous Events in Workflow Processes

Belinda M. Carter, University of Queensland, Brisbane, Australia; E-mail: bcarter@itee.uq.edu.au
Maria E. Orlowska, University of Queensland, Brisbane, Australia; E-mail: maria@itee.uq.edu.au

ABSTRACT

Exception handling during the execution of workflow processes is a frequently addressed topic in the literature. One important class of exceptions is those that represent predictable deviations from the normal behavior of the process that can be anticipated at design time. Such 'expected' exceptions are often caused by the occurrence of external events that are asynchronous with respect to the process. The desired exception handling response to these events will often depend on the current state of process execution. One important aspect of this state is the relevant process instance's progress through the process model, which can be expressed in terms of the set of currently executing tasks. In this paper, we present a qualitative discussion on issues relevant to the definition of policies for handling asynchronous, expected exceptions. First, we highlight the requirement for workflow control data to be referenced in the policies if these exceptions are to be handled in a meaningful way. We then demonstrate that the definition of exception handling policies is not a trivial exercise in the context of complex processes, and discuss correctness criteria for these definitions. Finally, we outline a methodology for policy definition to ensure that the policy for each event is complete and consistent with respect to all possible states of process execution for the relevant process model.

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 [4]. 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 [10]. 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 [6]. That is, workflows describe the 'normal behavior' of a process whereas expected exceptions model the 'occasional behavior'.

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 [6]. This asynchronicity makes it difficult to model exceptions with 'synchronous' constructs like tasks and flows, but since the exceptions are strongly associated with the application domain, they are part of the semantics of the process and so therefore should be incorporated within the process definition [4]. 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 relevant process instance. However, in most real world scenarios, reaction to the events will often depend on the state of the process instance in execution. While there are multiple aspects of the execution state of a process instance, we define 'state' as the 'stage of progression' of the process instance, as expressed through the set of currently executing tasks, for the purposes of this paper.

For example, consider a simple business process for processing customer orders consisting of sequential tasks *Receive Order*, *Approve*, *Pack*, *Dispatch*, and *Bill*. Suppose that the customer may cancel their order, provided that it is not ready for dispatch. Thus, the policy for this exception event consists of two rules: if the customer attempts to cancel the order after the pack activity is complete, the cancellation is to be rejected; and if the customer cancels their order before the pack activity is complete, compensation tasks are to be executed in order to perform a 'semantic undo' of the order.

The focus of this paper is on defining such policies for handling expected exceptions that are based on external events that occur asynchronously with respect to the process. Exceptional situations are usually very complicated [8] and we argue that it is easy to define policies that may produce unintended execution behavior. In the following sections, we present a brief introduction to the basic principles of workflow specification and execution, and then summarize the related work. We then describe the definition of exception handling policies, demonstrate the complexity in reasoning about policies defined over complex process models, and discuss relevant correctness issues. Finally, we outline a methodology for the definition of 'correct' policies with respect to all possible states of process execution. We conclude with an outlook for future research.

WORKFLOW SPECIFICATION AND EXECUTION

Before we can consider exception handling, let us first briefly summarize the basic principles of workflow specification and execution that are required for the subsequent discussion. Before a workflow process can be enacted, it must be specified. The process model describes the order of execution of tasks according to the business policies and resource/temporal constraints. Each task (or activity) is a logical unit of work within a process that may be either manual or automated but performed by a single workflow participant. In this paper, we will adopt graphical process modeling notation whereby rectangles represent forks and synchronizers (concurrent branching constructs) and ovals represent choices and merges (alternative branching constructs). A process *instance* is a particular occurrence of the process, for example, a particular order represents an instance of an order processing workflow.

A *workflow management system* (*WFMS*) is a system that completely defines, manages, and executes workflows. In this paper, we adopt the standard functionality for a WFMS, including states for activity execution, as presented in [14]. During process execution, the WFMS maintains internal *control data* that includes the internal state information associated with the various process and activity instances under execution. There are also two types of data that flows 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.

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 through a set of rules, thereby ensuring complete flexibility. For example [3] and [7] present approaches where the process is described through a set of *Event-Condition-Ac-*

tion (ECA) rules (c.f. [13]). 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. [13]). The importance of verification of the process model before deployment has been emphasized in [12]. Representing the entire process through sets of rules also makes it difficult to visualize the process, which is a drawback when it comes to validation and maintenance. Ultimately, since the process is validated and maintained by business domain experts, the process should be specified in an intuitive way. Therefore, as much of the process logic as possible should be represented in the graphical process model – that is, the entire core process, at a minimum, and ideally the exception handling functionality too.

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 [9] 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' [11]. The first approach is to implement exceptions through an explicit exception rule base [11]. Each exception is modeled 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 [4]. A different approach is presented in [10] 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 modeled as workflow processes themselves [11]. This approach is taken in [2], 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 execute in response to a particular 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 is noted that the set of states for a worklet-enabled process may be deduced by mining the process log file (c.f. [1]) but that full exploration of the specification of such conditions is yet to be completed.

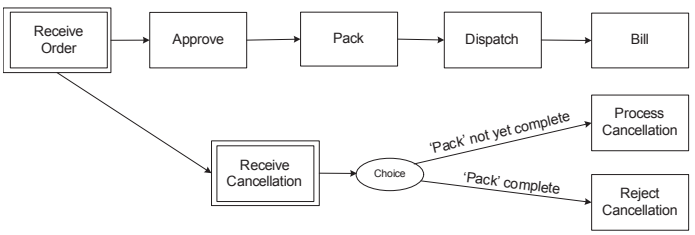
Another approach is to consider the exception handling processes as sub-processes within the core process. In the 'event node approach' discussed in [4], the workflow model includes a particular type of node, called an event node, which is able to observe asynchronous events and to activate its successor 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' [5], and to our knowledge, this issue has not yet been addressed in the literature. The definition of such conditions is the focus of this work.

EXCEPTION HANDLING POLICIES

An *Exception Handling Policy* ('*policy*') can be thought of as a set of ECA rules whereby for each such '*policy rule'* (*PR*), the *Action* is a sub-process ('exception handling fragment' in [6]) that is to be performed on observation of an *Event*, if the process execution state satisfies a particular *Condition*. The process execution state could involve many elements such as resource information and case data values, but for the purposes of this paper, we assume that the 'state' is described only through the set of currently executing tasks. The action may or may not involve terminating currently active tasks for that process instance, as dictated by the business requirements. The impact of the exception on the core process comprises the 'resolution' phase of the exception handling procedure, and while we acknowledge the importance of this phase, we focus our attention on the 'detection' and 'diagnosis' phases only in this paper (c.f. [11]).

While such policies could be defined and enforced through a set of rules, we emphasize that the underlying principles are the same if the exception handling is incorporated into the process model. We will adopt the event node approach for this discussion but it should be noted that these observations are generic and therefore applicable even if a worklet-style approach is adopted. For the sake of illustration, we distinguish a type of activity that assists with process coordination, called an *event listener*, which corresponds with the notion of an event node described in [4]. An event listener is an automated task that automatically completes on detection of a specific event. We emphasize the special role of these tasks in graphical process models by differentiating them with a two-line boundary. Event listeners allow the WFMS to observe relevant events asynchronously from the standard process in execution, enabling an immediate reaction to the event occurrence (following completion of the event listener task). This approach is also attractive because the exception handling functionality is incorporated into the process model but, due to the modular nature of the model, it is trivial to construct a view of the core process (or isolate a particular exception) for visualization purposes. It is also very extensible because the core process does not have to be modified. The example order process, incorporating the exception policy described earlier, is depicted in Figure 1. Observe the different process behavior on observation of the cancellation event according to the current state of the process instance.

Figure 1. Example scenario: Cancellation of customer order



Some of this exception handling process logic can be captured in the process model through the position of the event listeners – they can be placed at appropriate points such that if the event occurs before then, it is not observed by the process instance and therefore no action is taken. However, it will be usually be the case that a decision is (also) required to be made about how to handle the exception based on the workflow control data after it has occurred. To achieve this functionality (using standard modeling constructs, at least), a choice is required to be placed in the process model after the event listener activity to enforce the different PRs, where the choice conditions describe the status of the underlying process instance.

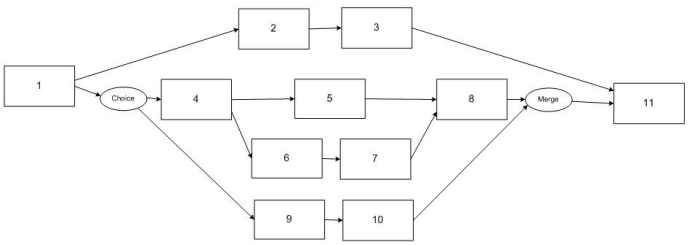
Generally, the conditions for choice constructs in a workflow process are based on case data that is generated during the activities that comprise the process (for example, an insurance application might undertake different treatment depending on whether it has been approved, with this decision being made during the execution of an activity). However, in order to enforce exception handling policies, workflow control data must also be referenced in the choice conditions. As already noted, this is a different type of data, maintained in the workflow log. From a specification point of view, there is no requirement that the choice conditions be defined on workflow relevant data only. We note that this reference to control data may impact on the underlying data model and system design, but we assume that such issues are resolved for the purposes of this paper.

CORRECTNESS ISSUES AND POLICY DEFINITION METHODOLOGY

We now consider the issue of correctness of exception handling policies. Note that we refer here to structural correctness rather than semantic correctness (that is, 'verification' rather than 'validation') since semantic validation depends on the particular application domain and so cannot be automated. We argue that there are two primary correctness criteria for exception handling policies – consistency and completeness. *Consistency* implies that no states have multiple actions defined for them, and *completeness* implies that there is an action defined for every possible execution state. If the exception handling behavior is incorporated into the process model then the choice conditions to be evaluated following the event listener must be mutually exclusive and collectively exhaustive, respectively, to satisfy these properties.

In graphical process models with no branches in the core process, each process instance is executing exactly one task at any point in time, and so these conditions are relatively simple to specify and verify. However, when concurrent branches are introduced into the process model, each process instance may be executing multiple activities at any point in time, and all combinations of these activities must be considered when defining policies. For example, for the process model

Figure 2. Example process model for discussion



depicted in Figure 2, the possible states 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)*.

In general, it is difficult to reason about the correctness of policies when the process contains a large number of activities and choice/split structures. Since a thorough consideration of all possible states for a process instance 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. For example, if one PR deals with the case where an event is observed 'before' reaching state *(3, 5, 6)* and another PR deals with the case where an event is observed 'during or after' reaching that state, the required behavior for the case where the instance state is *(2, 5, 6)* is not defined, and so the policy definition is not complete. Note that the intention during specification *may* be that no reaction is required in this case, but a case could 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 behavior is not required in particular situations. This requirement for completeness would help to ensure that the exception handling policies are an accurate and complete source of 'process knowledge', making it easier to understand, verify and ultimately maintain the policies.

The consistency of the policy should also be checked to ensure that if an exception event is observed, there is only one possible way of handling the exception. As for completeness, this property is not trivial to check in policies defined on complex processes. For example, a policy that is defined such that an action is performed if an event is observed 'after' state *(2, 4)* and another is performed if an event is observed 'during or before' *(3, 5, 7)* is inconsistent because there is a conflict for state *(3, 5, 6)* – both policies apply in this case, which may or may not have been intended.

In addition to being *complete* and *consistent*, PRs should be defined only for *valid states* of the process instance, otherwise the policy will be unnecessarily complicated with 'noise'. This 'simplicity' can be considered a secondary correctness criterion for exception handling policies. For example, the specification of a PR for *(2, 4, 9)* is also erroneous since this combination of activities is not a valid state due to the choice construct in the process model.

We argue that tools should be provided to either prevent errors from being introduced in the first place or to detect errors in the model before deployment, just as for the specification of the core process model. In the remainder of this section, we briefly propose a systematic method for the specification of policies in line with the former approach.

First, all tasks are assigned unique identifiers, and all instance sub-graphs are generated (c.f. [12]). The set of all possible states is then generated for each instance type, and the union of the sets is the set of possible states for the process model. Clearly, the set of states could be large – the cardinality is dependent on the number of forks in the model, branches for each fork, and activities on each branch. However, all cases must be considered in order to prevent erroneous policies being defined. Also, since this set is a feature of the process model, it can be reused to define policies for an arbitrary number of events, once it has been generated.

The set of states is then partitioned for each policy, with one PR (and action) per partition. We call the set of states in each partition the *scope* of a PR – that is, the set of states of execution for which the action associated with the PR is relevant. Thus, each PR has an event to be observed, a scope, and an action to be performed if the execution state at the time of event occurrence is contained within the set of states comprising the scope.

Once the policies have been defined, the method for enforcing them clearly depends on the exception handling approach that is adopted. However, if only event listeners and other standard workflow modeling constructs are employed, the event listener is immediately followed by a choice construct in the process model, and the scope of the policy corresponds to the choice construct condition that must be satisfied for the relevant sub-process to be executed. That is, each state corresponds to a logical disjunct in the choice condition that must be satisfied for the action associated with the policy to be performed. Upon observation of the event (and completion of the event listener activity), the condition expressions are evaluated using the current control data for the instance. In order to ensure completeness and consistency of the policy definition, every state must be a disjunct of exactly one condition for each exception event. The condition on one of the alternative branches will always be satisfied and the action (sub-process) associated with the relevant policy is then performed.

CONCLUSIONS AND FUTURE WORK

Although workflow exceptions occur infrequently, their handling should be automated whenever possible. In this paper, we discuss the definition of policies to handle expected, asynchronous exception events. The desired reaction in response to these events will often depend on the current state of process execution, and we argue that the definition of exception handling policies for complex processes is a challenging exercise. We have introduced correctness criteria for such policies and outlined a methodology for policy definition to ensure that the policy for each exception event is correct with respect to all possible states of process execution.

In our future work, we will develop a methodology for the automated verification of exception handling policies for any arbitrarily complex workflow model. We will also relax the restriction that state is described only through process instance position (currently executing tasks) and consider case data and 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 exception handling policies.

REFERENCES

[1] 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*, 47(2): 237-267.

[2] 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. *Proc. 17th Conference on Advanced Information Systems Engineering (CAiSE05) Forum*, June 2005, Porto, Portugal.

[3] 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.

[4] Casati, F. (1999) A discussion on approaches to handling exceptions in workflows. *SIGGROUP Bull.* 20, 3, 3-4.

[5] 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: 405–451.

[6] Casati, F., Pozzi, G. (1999) Modeling Exception Behaviors in Commercial Workflow Management Systems. *Proc. Fourth IFCIS International Conference on Cooperative Information Systems (CoopIS'99).* Edinburgh, Scotland.

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

[8] Luo, Z., Sheth, A., Kochut, K., and Miller, J. (2000) Exception handling in workflow systems. *Applied Intelligence*, Volume 13, Number 2, pp. 125-147.

[9] Mourão, H., Antunes, P. (2004) Exception Handling Through a Workflow. *Proc. 12th International Conference on Cooperative Information Systems (CoopIS'04)*: 37-54

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

[11] Sadiq, S. and Orlowska, M.E. (2000a) On Capturing Exceptions in Workflow Process Models. *Proc. 4th International Conference on Business Information Systems*. Poznan, Poland.
[12] 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.
[13] Widom, J. and Ceri, S. (1996) Active Database Systems. Morgan Kaufmann Publishers.
[14] Workflow Management Coalition. (1995) *The Workflow Reference Model*, Document Number TC00-1003, Issue 1.1.