Exception Handling Patterns for Hierarchical Scientific Workflows

Rafael Tolosana-Calahasan, José A. Bañares, Pedro Álvarez, Joaquín Ezpeleta
Department of Computer Science and Systems Engineering
University of Zaragoza, Spain
{rafaelt, banares, alvaper, ezpeleta}@unizar.es

Omer F. Rana
School of Computer Science
Cardiff University, UK
o.f.rana@cs.cardiff.ac.uk

ABSTRACT
Scientific workflows generally involve the distribution of tasks to distributed resources, which may exist in different administrative domains. Such a distribution may lead to faults that may arise at different levels: application level, enactment level, and resource management level, for instance. Detecting these faults, and subsequently adapting the structure of the workflow dynamically remains an important challenge. An approach to supporting such dynamic adaptation is presented, along with an evaluation of the approach using an example from the myexperiment.org workflow repository. An analysis of the overhead in using the approach is also presented, along with the benefits/pitfalls of using the proposed approach.

Categories and Subject Descriptors
C.4 [Performance of Systems]: Fault Tolerance

Keywords
Scientific Workflow, Exception Handling, Petri Nets

General Terms
Exception Handling in Workflows

1. INTRODUCTION
The enactment of a scientific workflow typically requires the mapping of tasks onto third party resources which are geographically dispersed. This distribution of tasks across multiple autonomous administrative domains makes the mapping process highly unreliable: exceptions may arise due to hardware and network failures, software faults and application errors. In some cases these failures are completely unpredictable and any recovery action is difficult. However, there are some failures whose nature, type and origin can be identified at system development/run time. Accordingly, workflow systems have adopted different fault tolerance mechanisms to make the enactment process more reliable.

We consider the exception handling process from an event-condition-action perspective. Consider $f_i$ being a single fault (hardware/software), and $\{f\}$ a set of faults, leading to a known event $e_i$. The event causes a single action $a_i$, or a set of actions (executed in some sequence) $\{a\}$ to be invoked. This can be expressed as: $(f_i,\{f\}) \rightarrow e_i \rightarrow (a_i,\{a\})$. Hence, we assume that the exception handling process relates to faults which can be detected, resulting in the generation of a particular event. Exceptions may be seen as unexpected events which occur during the enactment of a workflow. Such exceptions can occur at three levels: (i) application; (ii) workflow enactment engine and middleware; (iii) resource management/scheduling. Hence, the exception handling technique, in line with the notion of exceptions in the software engineering community [5], requires failures to be identified in advance and grouped into a set of failures related, for instance, by similarities of the conditions under which they might arise.

To support exception handling in workflow systems three requirements are significant. First, exception mechanisms should provide a separation of normal execution flows from the user-defined exceptional flows. It is important that normal execution flow is not impeded by additional control structures introduced to support exception handling. Second, as many scientific workflow systems currently support hierarchy: exceptions should be propagatable within the hierarchy of a workflow, until enacting an alternative action is more suitable. Third, considering the inherent dynamism of distributed environments, users should be allowed to specify different exceptional flows to handle specific failures based on the fault context. According to such context, the most suitable action will be chosen and loaded dynamically, so that the workflow system can be dynamically adapted [3].

Even though there are many scientific workflow systems, very few of these support exception handling techniques and only Hoheisel [2] addressed and tackled the problem in hierarchical scientific workflows with the required level of dynamism. A static Petri net formalism is used in [2], thereby
limiting its use to pre-defined exception handling scenarios. We propose a technique based on the three previous requirements for handling exceptions which allow workflows to re-configure and to adapt themselves in case of failure. We integrate and implement this technique in DVega’s workflow system [7]. Instead of the subclass of Petri nets that Hoheisel uses, we utilise the Petri net subclass of Reference nets, which (due to their dynamic nature) are much better suited for workflow adaptation. Furthermore, we formulate the technique in terms of workflow patterns: starting from a simple pattern for defining hierarchical workflows, we then extend this basic pattern in two ways, one for propagating faults within a workflow hierarchy and another for responding to an exception at a particular workflow level. This treatment consists of alternatively emitting the most suitable subprocess workflow among the alternative candidates depending on the execution context. We demonstrate our approach on a real hierarchical workflow example taken from the myexperiment.org project.

The rest of the paper is organised as follows, in Section 2, a brief background knowledge about Petri nets and their role in workflows is given. Section 2.1 shows DVega’s workflow language extension for supporting hierarchical workflows and a workflow pattern for expressing hierarchical workflows is defined. In Section 3, related work is reviewed briefly. In Section 4, two patterns for dealing with exceptions in hierarchical workflows are defined and their mechanisms explained. In Section 5, the patterns are applied to a real workflow example taken from myexperiment.org. Afterwards, a brief analysis of the overhead generated by the approach is given. Finally, the conclusions and future work are provided.

2. BACKGROUND

An ordinary Petri net can be defined informally as a bipartite directed graph which consists of places, transitions, arcs and tokens (see [4] for a formal definition). There are many extensions to ordinary Petri nets which provide higher levels of abstraction and improve the modelling potential of ordinary Petri nets. We use a specific type of High-level Petri net, Renew’s Reference nets. Renew 1 is a Reference net interpreter and a Reference net graphical modelling tool used in this work. Renew’s Reference nets are a special subtype of Net-within-Nets [8]. The characteristics and behaviour of ordinary Petri nets are also present in Reference nets, but they provide other useful extensions: (i) Reference net tokens can also be a Java object or another Reference net, and all of these nets can communicate with each other by means of synchronous channels. A synchronous channel can be established only between two nets and it allows a net to send (invoke) a message to another net in a synchronous way. (ii) Reference net transitions can be equipped with a variety of inscriptions such as expressions, actions or guards. Expression and action inscriptions are ordinary expressions evaluated while the net simulator searches for a binding of the transition, the result of this evaluation can be used to influence the binding of variables that are used elsewhere. Guard inscriptions are expressions that are prefixed with the reserved word guard, so that a transition can only be fired when all of its guard inscriptions evaluate to true. (iii) Reference net arcs can also have inscriptions. By default, an arc without inscription transports a black token (the token of ordinary Petri nets), but when arcs have variables as inscriptions, they transport Java objects or other nets. (iv) Renew’s Reference nets are dynamic, and in this way differ from other High-level Petri net approaches. This dynamism is achieved by means of the new construct that can be part of an expression inscription at a transition, as a result a new instance of an object net can be created at execution time. This feature totally overcomes the static nature of other high-level Petri nets. Renew’s Reference net domain is formed by the set of all possible tokens, including any Java variables or objects and any Reference net instances.

2.1 Hierarchical workflows in DVega

DVega’s workflow language is based on Reference nets and orchestrates workflow nodes by means of well-known control structures [6] namely sequence, parallelism, join, choice and iteration. Other more complex control patterns can also be modelled with Petri nets when required, as proposed by [6]. We extend DVega’s workflow language for expressing hierarchical workflows and dealing with exceptions with different levels of granularity.

Initially, DVega’s workflow nodes were simple tasks, we propose that workflow nodes can also be subprocess workflows and these subprocess workflows, at the same time, can also handle either tasks or other subprocess workflows recursively, so that users can express hierarchical workflows with different levels of granularity. Both types of nodes can be implemented by the same Petri net construct, a workflow node pattern that is depicted in Figure 1. The main idea behind this construct is that it receives the input data (variable args in the figure), then, at this time, it dynamically creates a new net instance of a net type in Transition t1 (in the figure, the net type is called WorkflowModel) and a new instance of it is obtained by the action w: new WorkflowModel which is going to be enacted at Place P1. It is achieved by means of synchronous channels (see w:begin(args) and w:end(result) in the figure). Once the enactment is finished, the result will be obtained in Transition t2 and will be passed to another workflow node. The net type created will therefore determine whether the workflow node corresponds to a simple task or a subprocess workflow.

3. RELATED WORK

Hoheisel [2] proposes a workflow system for supporting hierarchical workflows and handling exceptions whose workflow language is based on Hierarchical Petri nets (HPNs). HPNs places and transitions within a Petri net to be refined with additional Petri nets, thereby facilitating the modeling of large real-world systems. However, the formalism, as
defined in [1], may not be suitable for the dynamic environment of scientific workflows, because the structure of these nets is inherently static. For instance, there is no construct in HPNs for properly meeting the requirements proposed in section 1, by which users are allowed to specify different exceptional flows, allowing the most suitable one to be chosen and loaded dynamically. Moreover, HPNs also do not provide any mechanism by which exceptions can be propagated within the formalism. Hoheisel solves the problem by building a HPN parser. Our proposal, in contrast, is based on Reference nets, which provide mechanisms inside the formalism for modelling all of these issues thereby allowing us to express a solution using the formalism.

A review of fault tolerance mechanisms in scientific workflow systems can be found in [9]. Due to space limitations we re-view fault tolerance mechanisms in Taverna & Triana (two popular workflow systems). In Taverna 2, two techniques are provided to the user for dealing with faults: task retry and alternative task. With task retry, workflow designers are allowed to indicate the maximum number of times that a task will be retried in case of execution failure. This can also be applied to subworkflows whenever they fail. The alternative task technique allows users to specify a different task in case of failure, after a maximum number of retries have been attempted. However, alternative subworkflows cannot be specified. Triana’s 3 support for fault tolerance is generally user driven. For example, faults will generally cause workflow execution to halt, display a warning or dialog, and allow the user to modify the workflow before continuing execution. At workflow level light-weight checkpointing and the restart or selection of workflow management services are currently supported. At the middleware and task levels, all the listed faults can be detected by the Engine or GAT, except for deadlock, livelock and memory leaks. At the lowest level, machine crashes and network errors are recognized by GridLab GAT and the Triana Engine respectively, but recovering from these faults or preventing them is only planned for future versions.

4. EXCEPTION HANDLING PATTERNS
Whenever an exception is raised in hierarchical workflows, considering different levels of granularity instead of just task level may be useful under certain circumstances. Indeed, users may have different subworkflow specifications available for a certain computation rather than just simple alternative tasks. For example, a user might have two subworkflows, one of them may run faster but might be more unreliable, whereas the second one is likely to run much slower but is reliable. In this case, the user can use the faster subworkflow as the normal execution flow and specify the second one just as an alternative subworkflow in case of failure.

On the other hand, when an exception is raised at the lowest levels of the workflow hierarchy, that is, at task level, users will have to decide the suitable level of granularity for enacting an alternative flow and, once decided, the exception will have to be propagated up in the hierarchy. Users will also have to design the alternative subworkflows for the exception. There may be many alternative subworkflows for a given workflow pattern. Figure 2: Exception Propagation Pattern

4.1 Exception Propagation Pattern
In DVega, all workflow node types are implemented by the same pattern, shown in Figure 1. This pattern expects nodes to terminate their execution in the normal way, without any failure. Figure 2 shows an extended version of the original workflow node pattern for capturing exceptions and propagating them through a workflow hierarchy. It should be noticed that it has two more transitions and a place. As in the original pattern, the node starts its normal execution in Transition t1 and ends its normal execution at Transition t2. However, the execution can finish abnormally and this is represented by Transition e1. It should be noticed that there is no indeterminism between Transitions t2 and e1: Transition t2 will only be fired in case of normal enactment, whereas Transition e1 will only be fired in case of an exception and the different synchronous channels at both transitions will enable one or another alternatively. Synchronous channel v:exception(ex) at Transition e1 captures the exception from a lower level as well as its cause (Variable ex). The role of Transition e2 at this pattern is also important as it propagates the exception to upper levels by means of Synchronous channel :exception.

4.2 Exception Treatment Pattern
Figure 3 shows a pattern for capturing an exception from lower levels and treating it at this level by applying alternative actions. The pattern is also an extension of the original workflow node pattern. It receives as an input the arguments required for the enactment and a named list of subworkflow alternatives in case of failure. As in the exception propagation pattern, there is no indeterminism between Transitions t2 and e1. When an exception is raised at a lower level, Transition e1 is enabled and fired, Variable ex will retrieve the exception cause, obtained by the synchronous channel. In Transition e2, the best candidate of the alternative subworkflows should be chosen by using a policy: in this case the first candidate from the list of alternative subworkflows is selected (though more complex rules for selection may also be used). Then a new candidate instance is created (action nw=WFName(AltWf)). In Transition e3, the enactment of the alternative subworkflow is initiated (by means of the subworkflow’s initial synchronous channel: v:begin). The arrows going from the pattern to the nets (subworkflows) on the right represent synchronous channels which allow the recovering from an exception, but, in this work, we assume that the exception context will lead to only one alternative subworkflow. Two patterns are proposed for accomplishing these actions: the exception propagation pattern and the exception treatment pattern.

2http://taaverna.sourceforge.net/
3http://www.trianacode.org/
pattern to enact different subworkflows – on the front the subworkflow at normal execution is schematised and behind it in dotted lines the alternative candidates.

Both patterns could be easily combined together in a unique pattern. Thus, in case all the subworkflows on the alternative list fail, the exception can be propagated up in the hierarchy, preventing the system from entering in an endless loop due to a persistent fault.

5. MODELLING EXAMPLE

Figure 4 represents an example of a two-level hierarchical workflow, taken from myexperiment ⁴ project. This workflow takes four inputs: query_string, document_index, search_field and maxHits. The last three of these inputs are submitted to a database for finding documents up to maxHits. The first input, query_string is however optimized through a Biooptimize_query subworkflow by extending the original query with an additional string that enables the output to be ranked based on the year of publication. This is achieved through the Lucene_year_priorities service. We use our approach to represent this workflow using Petri nets, because it has relevance in a number of scientific domains (not just BioSciences, for which it has currently been developed) – where searching for documents in publications repositories is an important requirement.

Figure 5 shows a hierarchical-structure equivalent model in DVega’s workflow language, but, in this case, we integrated the patterns defined in Section 4 in order to cope with exceptions at different levels of granularity. Figure 5 represents the top hierarchy level and consists of a sequence of two workflow node patterns. The first in the sequence, a node with an exception treatment pattern, enacts Sub-workflow Biooptimize query, whereas the second pattern, a node with an exception propagation pattern, enacts Sub-workflow Retrieve. In Figure 6, the details of Subworkflow Bio_optimize query are detailed. It consists of just a simple task called Prioritize lucene query, represented by a workflow node that follows the exception propagation pattern. Therefore, whenever this simple task fails, an exception is caught in Transition e₁ of Figure 6 and then thrown up in the hierarchy by Transition e₂. Up in the top of the hierarchy, the exception will be caught by Transition e₁ of Figure 5. Then, in Transition e₂ the first candidate of the alternative subworkflow recover list will be taken. This subworkflow will be created dynamically and its execution started at Transition e₃. Subworkflow Retrieve can be modelled in an analogous way and it is not shown here for space limitations.

6. OVERHEAD ANALYSIS

In our proposal for managing exceptions, both the treatment exception pattern and the exception propagation pattern impose an overhead on the execution time of workflows. The overhead of the treatment exception pattern is generated by the enactment of Transitions e₁, e₂, e₃ and their actions (see Figure 3). This overhead mainly depends on two factors: (i) the selection of an alternative subworkflow (Transition e₂), achieved by taking the first element of a list of names containing candidate subworkflows. This action can be undertaken in constant time; (ii) the instantiation of the selected alternative subworkflow (Transition e₃), corresponding to action nv:new Wf4Name(AltWF) (see Figure 3) which dynamically loads and creates an instance of a subworkflow. An experiment on the exception treatment pattern was accomplished in order to analyse the scalability of this pattern. Different sized workflows (ranging from 1 task up to 1000 tasks) were loaded and enacted with the Renew tool. The result of this experiment is depicted in Figure 7: showing the subworkflow size, measured in the number of tasks, with the time overhead imposed by using the pattern, measured in milliseconds. According to the measurements, it can be stated that there is a strong linear correlation between both variables (R² = 0.9) and the minimum overhead obtained is for the 1-task subworkflow (10.70 ms), whereas the maximum overhead (341.80 ms) corresponds to the scenario of the 1000-task subworkflow. We attribute the anomalous behaviour in execution time overhead (from task count of 400 to 600, compared to an increase from 600 to 800) due to the way in which the garbage collector of the JVM is used by Renew. However, this variation is not significant, as indicated by the R² statistical test shown in Figure 7. A similar argument can be made for Figure 8, where synchronisation between transitions in the Petri net is needed to support the propagation of an exception across multiple workflow levels in the hierarchy.

The overhead of the propagation pattern is generated by the propagation of the exception from its origin in a certain level of the hierarchy, until the level in which it is being treated. It should be noticed that, in order to propagate an exception, a sequence of propagation patterns must be concatenated (each propagation pattern belongs to a different level). Thus, the overhead of this pattern will depend on the length of the chain of patterns and in the communication and synchronisation between them via synchronous
Figure 3: Exception Treatment Pattern

Figure 5: Workflow Retrieve Biodocuments, modelled with DVega’s workflow language (top hierarchy)

channels. These communication and synchronisation actions are accomplished by Transitions e1, e2 at each propagation pattern (see Figure 2). An experiment was carried out in Renew in order to measure the scalability of this pattern. Up to 10 levels of hierarchy were considered to demonstrate the effectiveness of the approach. Although 10 levels of hierarchy may be too large a number for modelling a hierarchical workflow. Figure 8 shows the results of the experiment: the level of hierarchy (from 1 up to 10) vs. the execution time overhead due to the propagation of the exception through the levels. There also exists a linear correlation between both variables ($R^2=0.9$) and the minimum (1.70 ms) and the maximum (16.20 ms) correspond to 1 level of propagation and to 10 levels of propagation, respectively. Therefore, both experiments show that the proposal is scalable.

The experiments were undertaken with a hardware 2.2 GHz Intel Core 2 Duo processor, 2.5GB 667MHz DDR2 SDRAM and 4MB L2 cache, whereas the software used was Renew 2.1, Java VM 1.5.0_13 and Mac OS X 10.5.4.

7. CONCLUSIONS

Exception handling in scientific workflow remains an important challenge. Dynamically adapting an existing workflow when such an exception is detected is often missing from many existing workflow systems. Hoheisel [2] recognized

Figure 7: Overhead of the Exception Treatment Pattern
this and proposed a Petri net-based representation for managing such exceptions. However, the proposed model utilized a static Petri net formalism which is difficult to adapt dynamically. The approach proposed in this paper utilizes Reference nets – a particular Petri net formalism that enables dynamic creation of new Petri nets, providing the abstractions to support hierarchical exceptions and dynamic adaptability of workflows in an easy way – and the Renew tool to execute Reference nets. Specifically, we demonstrate how dynamic adaptation could be supported through two design patterns which were integrated and implemented in the DVega workflow system: (i) an exception treatment; and (ii) an exception propagation, pattern. Our approach enables exceptions, captured at different levels of a workflow, to be propagated through the workflow hierarchy to ensure that they can be handled at the most appropriate level.

Additionally, we demonstrate how the approach can be used for a realistic workflow obtained from the myexperiment.org project, and provide experimental results for the overhead analysis in dynamically replacing a subworkflow. Our focus in doing this is demonstrating that our approach does not just provide a model of the workflow, but is an executable workflow that can be used by end users. We also demonstrate how a workflow intended for use in the Taverna workflow engine can be represented using Reference nets, and used to achieve the same outcome. Although we appreciate that a Petri net-based formalism is more complex (in terms of representation) than a task graph representation in Taverna – we believe that the added benefits of supporting exception handling and dynamic creation of subworkflows overcomes these limitations.

Acknowledgement This work was supported by the research project TIN2006-13301 by the Spanish Ministry of Education and Science. Rafael Toloza’s work was supported by “Programa Europa XXI de Estancias de Investigación” ref. IT 1/08 by DGA (CONAID) and CAI.

8. REFERENCES