CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2008; **00**:1–7

Adaptive Exception Handling for Scientific Workflows

Rafael Tolosana-Calasanz[1,*,†], José A. Bañares[1], Omer F. Rana[2], Pedro Álvarez[1], Joaquín Ezpeleta[1], Andreas Hoheisel[3]

[1] *Instituto de Investigación en Ingeniería de Aragón (I3A). Department of Computer Science and Systems Engineering. University of Zaragoza, Spain*
[2] *School of Computer Science. Cardiff University. Wales, United Kingdom*
[3] *Fraunhofer Institute for Computer Architecture and Software Technology. Berlin, Germany*

SUMMARY

Scientific workflow systems often operate in highly unreliable, heterogeneous and dynamic environments, and have accordingly incorporated different fault tolerance techniques. We propose an exception handling mechanism, based on techniques adopted in programming languages, for modifying at run-time the structure of a workflow. In contrast to other proposals which achieve the required flexibility by means of the infrastructure, our proposal expresses the exception handling mechanism within the workflow language – primarily as two exception handling patterns which are exclusively based on the Reference Nets-within-Nets formalism (a specific type of Petri nets). Therefore, workflow users can have better control and understanding of the behaviour of their workflow without having to be aware of the underlying infrastructure.

keywords: Scientific Workflows, Exception Handling, Petri nets

1. INTRODUCTION

A key characteristic of scientific workflows (which differentiate them from workflows in the business domain) is that they need to have a flexible design, and support exploration – rather than be prescriptive (as in business computing) [1]. A scientist may be interested in exploring a parameter space, or interested in undertaking multiple *what-if* scenarios, before converging on suitable parameters to define an experiment. This may involve an exploration of variants,

*Correspondence to: Departmento de Informática e Ingeniería de Sistemas
Universidad de Zaragoza
María de Luna s/n 50018 Zaragoza - Spain
†E-mail: rafaelt@unizar.es
Contract/grant sponsor: Spanish Ministry of Education and Science; contract/grant number: TIN2006-13301
Contract/grant sponsor: DGA (CONAID) and CAI. Programa Europa XXI; contract/grant number: IT 1/08

and manipulation of different workflow configurations – often leading to significant changes to a workflow as the experiment evolves to some *useful* outcome. We may therefore differentiate between an "abstract" workflow – which describes an experimental process the scientist is considering, and a "concrete" workflow – which is an engineered realisation of the process. Ideally, it must be possible to re-execute a concrete workflow to obtain the same experimental outcome every time. However, as the computational infrastructure evolves, this requirement is often difficult to achieve in practice.

As concrete workflows intended for enacting large and complex scientific activities require the mapping of tasks onto third party, heterogeneous, stand-alone and distributed resources. These characteristics lead to faults which, in some cases, are completely unpredictable and any recovery action is difficult. However, there are some faults whose nature, type and origin can be identified at system development/runtime. Accordingly, in order to make the enactment process more reliable, scientific workflow systems have incorporated different fault tolerance mechanisms. The exception handling technique can be used as a fault tolerance mechanism and can treat a subset of these faults.

An exception can be seen as an unusual event, erroneous or not, that is detectable either by hardware or software and that may require special processing. Thus, consider a single fault (hardware/software) f_i, 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: $(f_i|\{f\}) \rightarrow e_i \rightarrow (a_i|\{a\})$.

Exception handling mechanisms have been widely used in programming languages in order to give the programmer means to adapt the behaviour of operations, as well as to separate the failure semantics from the program logic, facilitating the design of readable and comprehensible code. Additionally, general purpose workflow systems have also integrated these mechanisms for improving reliability. However, due to the specific nature of scientific workflows, further research is required in order to integrate exception handling into scientific workflow systems. Scientific workflows also generally have a nested hierarchical structure composed of smaller integrated sub-workflows acting as software components. This is the approach followed by workflows made available at the `myexperiment.org` repository. In `myexperiment.org`, scientists can publish their own workflows, but they can also use and integrate other scientists' workflows into their designs, thus forming bigger hierarchical nested structures. Furthermore, scientists may find different sub-workflow components there that, having the same signature – input and output I/O interfaces, can alternately provide the same functionality. This characteristic could be exploited in case of failure of a sub-workflow component. Thus, the failed sub-workflow signals (notifies) an exception, which can lead to it being aborted and dynamically replaced by a suitable alternative candidate. However, the system has to support a high level of dynamism for providing such a functionality. Among the existing scientific workflow systems, very few support exception handling, among which Hoheisel [2] addressed the problem in hierarchical scientific workflows and recognised the dynamism required. However, his approach uses an ordinary Petri net formalism for representing workflows and integrates the exception handling mechanism in them, thereby limiting its coordinated use to pre-defined exception handling scenarios. Although his approach supports dynamic modifications of the workflow structure during workflow run-time, the

modifications are not clearly modelled within the formalism and, in consequence, the required dynamism is achieved by means of the infrastructure.

To support exception handling in workflow systems, we have identified four basic requirements. 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. Therefore, the complexity of the normal execution flow is not increased by the exception handling, and reusability of sub-workflow components is not also diminished either. 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 (sub-workflow) will be chosen and loaded dynamically, so that the workflow system can be dynamically adapted [3]. Four, workflow languages should be extended with the appropriate abstractions to express the three previous requirements. Thus, workflow designers can have a better control of the workflow behaviour. Besides, workflow interpreters/enactors should be able to interpret these abstractions correctly.

In this paper, we propose a technique for exception handling in hierarchical scientific workflows with the required level of dynamism, in the context of the previous requirements. We assume here that the exception has been detected properly and is ready to be handled by the system. We integrate this technique in DVega's scientific workflow system [4]. Instead of the subclass of Petri nets that Hoheisel uses within the Grid Workflow Execution Service (GWES), we utilise the Petri net subclass of Reference nets, which (due to their dynamic nature) are much better suited for workflow adaptation – by allowing new Petri net structures to be generated dynamically (generally triggered by the passing of tokens). It is also possible to support guards and annotations in Reference Nets that enable Petri net execution to be more effectively managed. Additionally, the mechanism is completely designed by using the Reference net formalism with no additional extensions. We formulate the mechanism in terms of workflow patterns: one for propagating exceptions within a workflow hierarchy and another for aborting and replacing a failed sub-workflow component at a particular level. This treatment consists of alternatively enacting the most suitable sub-workflow among the alternative candidates depending on a selection policy. We propose three different selection policies: Event-Condition-Action (ECA) rules considering the failure context, a policy based on choosing the candidate with the least execution time estimation and a policy based on choosing the candidate with the lowest cost estimation. Finally, we demonstrate our approach on a real hierarchical workflow example taken from the `myexperiment.org` project. In summary, a workflow can be represented conceptually using Reference Nets (a type of Petri net formalism), and can be simulated using the Renew tool. A Reference Net can then be enacted (executed) through DVega (as the workflow enactment engine). We also demonstrate how a workflow expressed as a graph (for execution through Taverna) can be executed using DVega – thereby demonstrating that existing workflows can also be supported through our approach.

The rest of the paper is organised as follows: in Section 2, a brief overview of Petri nets and their role in workflow is given. In Section 2.2, we show how DVega's workflow language and its extension for supporting hierarchical workflows (using a *pattern* for expressing hierarchical

workflows) is given. In Section 3, the types of exception that a scientific workflow system may have to address are presented. In Section 4, our proposal for exception handling is described and 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`. In Section 6, a brief analysis of the overhead generated by the pattern is given. Section 7 discusses and compares previous related work. Finally, the conclusions are provided.

2. BACKGROUND

2.1. Petri nets

An ordinary Petri net can be defined informally as a bipartite directed graph which consists of places, transitions, arcs and tokens (see [5] for a formal definition and a general introduction to the formalism). There are many extensions to ordinary Petri nets – such as High-level Petri nets and timed Petri nets – which provide many additional features such as higher levels of abstraction. Ordinary Petri nets and their extensions have been widely used for the specification, analysis and implementation of workflows [6]. In the scientific workflow community, they have also been utilised and GWorkflowDL [7, 8], Grid-Flow [9] and FlowManager [10] are representative examples of this.

In this work, we use a specific type of High-level Petri nets, Reference Nets-within-Nets [11] (or just Reference nets), whose name is due to the fact that their tokens can also be nets. Therefore, nets can have other nets inside, forming a nested structure. Nevertheless, the main distinctive aspect of Reference nets is their dynamic nature: in Reference nets, a net itself can express the creation of new net-typed token instances explicitly. Furthermore, a parent net (known as *system net*) and its token child nets (known as *object nets*) can interact with each other via a communication mechanism called *synchronous channels*. Intuitively, this kind of communication is based on the synchronisation between a transition in a system net with a transition in an object net. Besides, a channel can also house variables whose binding is based on unification, thereby providing a flexible and bi-directional communication mechanism. In synchronous channels, one of the communicating nets acts as an invoker, trying to synchronise with another transition, while the other peer acts as an invokee. For example, the synchronous channel expression at the invoker net side `netexpr:channelname(expr, expr, ...)` indicates that `netexpr` is an expression that must evaluate to a net which must have a transition with a synchronous channel called `channelname`. At the invokee side, the expression should be of the form `:channelname(expr, expr, ...)`, completely ignoring whoever the invoker can be. Subsequently, only when the unification of channel variables is possible and the corresponding and implied transitions are enabled, these transitions can be fired (the firing is simultaneous) and a bi-directional communication can be established. These aspects along with the inherent dynamism of the formalism provide a key communication mechanism for supporting environment adaptability of workflows.

In order to illustrate the main concepts of Reference nets, let us consider a scenario involving a `Resource Consumer` and two `Resource Providers`, where `Provider 1` and `Provider 2`

can simultaneously allocate two resources. Figure 1 depicts a Reference net that models this scenario. The consumer can request a particular service with a required level of Quality of Service (QoS) to any of the two providers. In the model, the consumer accomplishes the request via Synchronous channel `match(request,QoS,resource)`. Variables `request` and `QoS` are bound by the consumer to a service name and a QoS level, respectively, whereas Variable `resource` will be bound with a resource by the provider that matches the request.

The unification of these channel variables (`request`, `QoS` and `resource`) between the consumer and the providers gives the following possible firing modes:

- Synchronized firing modes of `Consumer` with `Resource Provider 1`:

```
{request=serv1,QoS=2,inputData=data1}
{service=serv1,QoS=2}
and
{request=serv2,QoS=2,inputData=data2}
{service=serv2,QoS=2}
```

- Synchronized firing modes of `Consumer` with `Resource Provider 2`:

```
{request=serv2,QoS=2,inputData=data2}
{service=serv3,QoS=2}
 and
{request=serv3,QoS=2,inputData=data3}
{service=serv3,QoS=2}
```

Once a provider accepts the request and allocates (`res:allocate()`) the resource, the consumer can start the execution of the required service. The Consumer can execute the service: the resource is provided the input data (`resource:begin(inputData)`) and after the execution, the output data is provided to the consumer (`:end("outputData")`). At this point, the provider frees the resource (`res:free()`) and can create new resource instances when required (`res:new resource`).

Another of the advantages of the Reference Net formalism is that there exists a tool that can interpret them: Renew[†] [12]. Renew is a Java-based Reference net interpreter and a Reference net graphical modelling tool which was used in this work. Renew also utilises tuples and Java expressions as the primary inscription language. In addition to object nets, transition firings can also transport Java objects. Transition inscriptions which are prefixed with the reserved word `guard` have constraints on their potential firing modes. Additionally, transition inscriptions that are prefixed with the reserved word `action` represent atomic actions that may be executed after the firing of the transition.

Reference nets and the core of the Renew interpreter have been utilised in [13] for implementing and enacting Web-based interaction in a service-oriented system. DVega [4],

[†] `http://www.renew.de`

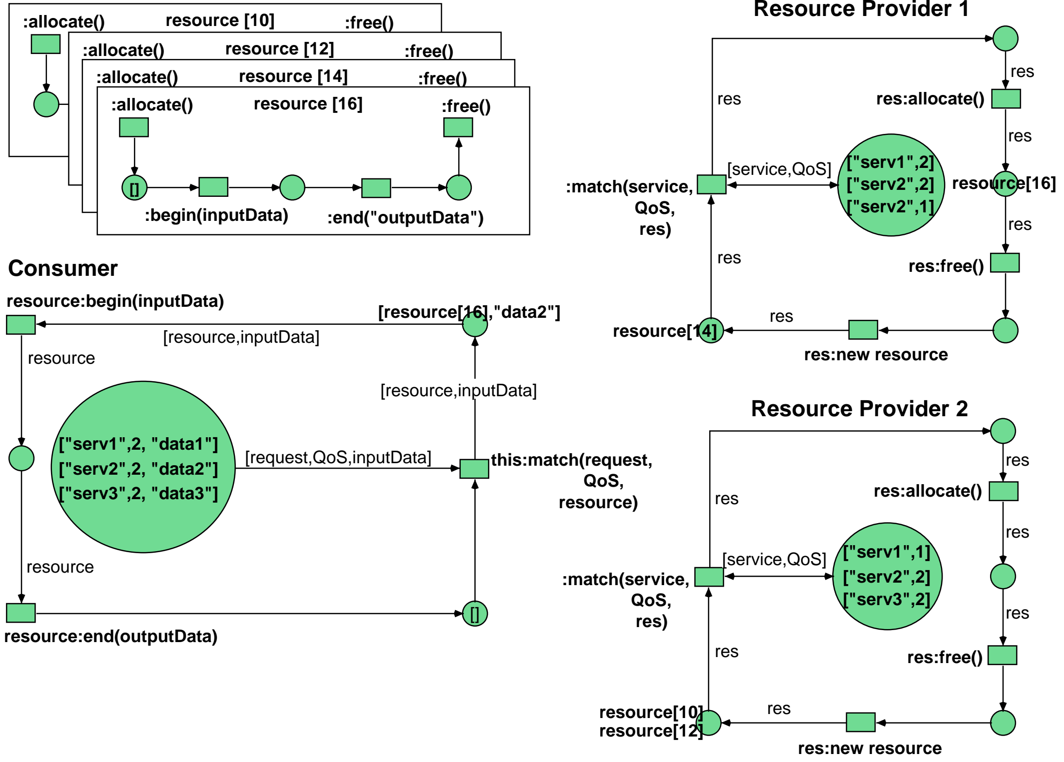

Figure 1. A Renew net model that represents a `Service Consumer` and two `Resource Providers`. References to `Resources` net instances are managed by consumers and providers.

a service-oriented Grid workflow management system, was also implemented by means of Reference nets. DVega consists of a set of loosely coupled services co-operating with each other. In DVega, the execution flow of its services is isolated from their interactions and these interactions are explicitly modelled and can be dynamically interpreted at run-time, providing a high degree of flexibility when interacting with the environment.

2.2. Hierarchical scientific workflows in DVega

DVega's workflow language is based on Reference nets and orchestrates workflow nodes by means of well-known control structures [14] namely sequence, parallelism, join, choice and iteration. Other more complex control patterns can also be modelled with Petri nets when required, as proposed by [14]. We extend DVega's workflow language for expressing hierarchical workflows and dealing with exceptions with different levels of granularity.

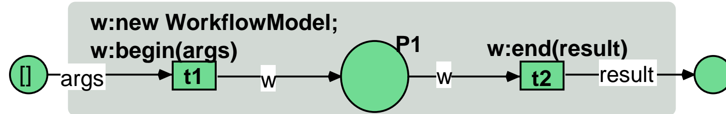

Figure 2. Workflow Node Pattern at DVega's workflow language

Initially, DVega's workflow nodes were simple tasks. In this work, workflow nodes can also be sub-workflows and these sub-workflows, at the same time, can also handle either tasks or other sub-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 2. 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 sub-workflow.

Figure 3 shows a schema of a hierarchical workflow modelled with DVega's workflow language. In each level of the hierarchy, several workflow nodes can be seen. At the top of the hierarchy, one of the workflow nodes has a sub-workflow being enacted and by means of the `begin` and `end` synchronous channels the workflow node sends the arguments required and obtains the results. At the same time, this middle sub-workflow has its own workflow nodes which handle simple tasks representing the bottom of the hierarchy. Once again as before, the communication is established by means of synchronous channels.

3. EXCEPTION TYPES

According to [15], three architectural levels, common to many Grid workflow management systems, can be identified: the *workflow level* – this is the top level, where workflow specifications are enacted by utilising internal components and external services from lower architectural layers, the *middleware level* – in general terms, it provides specific services that may facilitate workflow enactment activities and also gives access support to available resources, and the *resource level* – the lowest level, formed by the set of heterogeneous and distributed devices and machines that can actually execute workflow tasks.

Hence, depending on the exception produced, three main exception types for scientific workflows can be identified:

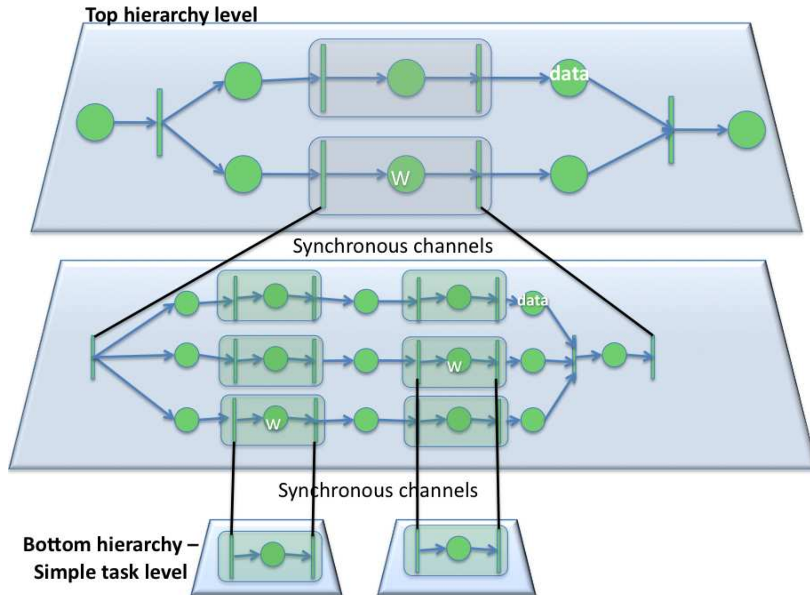

Figure 3. Hierarchical Workflow Schema Example in DVega's workflow language

- *Workflow-level Exception Type*: at workflow-level, the exceptions that can arise are related to non-satisfied assertions or deadline expirations. Each workflow node has an input assertion and an output assertion. When an input assertion of a workflow node is not satisfied (i.e. a file name was given as an input data and the file was not found), an exception is generated. Non-satisfied output assertions happen due to the inability of the workflow node to produce the output data and also produce exceptions, but in general terms, their detection is more complex than in the case of input assertions. Additionally, in some other situations, users may also define other types of assertions in the workflow over elements in the control-flow, such as the data. Users may also impose deadline expirations for specific tasks or even sub-workflows. In most cases, non-satisfied assertion are detected by monitoring and evaluating the assertions.
- *Middleware-level Exception Type*: workflow systems usually utilise intermediary middleware service for interacting with autonomous, external resources. In the heterogeneous and highly dynamic environments of scientific workflows, middleware platforms have to deal with many different failures and errors such as network failures, resource unavailabilities (task allocation can arise at distribution time – no resource can be found which meets the specified allocation criteria – or at some time after allocation) or middleware software errors.

Many middleware frameworks feature different fault tolerance techniques that try to mask failures and errors at the workflow-level. For instance, some middleware frameworks provide meta-scheduler services that, in case of resource failure, can retry the job (in the same or an alternative resource). However, there may be certain circumstances where these masking mechanisms are not successful. Furthermore, the detection of the fault (or error) may be difficult, specially in case the incident was completely unexpected. In both cases, the problem has to be addressed at the workflow-level and exception handling represents a good technique for dealing with these failures and errors because of the reasons given in Section 1. The detection of the anomalous situation is the first required step. Sometimes the middleware itself may inform about the incident, otherwise the only sign of the abnormality is the inability of a single task (or sub-workflow), during workflow enactment, of producing the required output data.

- *Resource-level Exception Type*: in some cases, workflow systems use middleware support just to directly utilise a resource and no intermediary masking fault tolerance mechanism is involved. In this case, workflow systems have to deal with the incidents that may arise at the resources. There are many possible different incidents: again the unavailability of a resource, software errors and also other problems such as local-scheduling incidents, memory leaks, operating system errors, hardware faults, network failures, etc. In many cases, they may be well suited for resolution via exception handling.

These types of exceptions, if properly detected, can be handled with the technique proposed in the next sections.

4. EXCEPTION HANDLING

The exception mechanisms proposed in this paper and incorporated in DVega's workflow language are based on Goodenough's [16] proposal for programming language and on Yemini's [17] replacement model, a modular mechanism for dealing with exceptions in modular programming languages. The imported idea from the replacement model to a workflow language is that a failed element can be dynamically replaced by another one at run-time.

4.1. DVega's Exception Handling Mechanism

A workflow in DVega is a composite of workflows (called sub-workflows) and simple tasks. It can be seen as a hierarchical structure with two different types of workflow nodes: the simple task node, that is, a node with just one descendant – a simple task – and the sub-workflow node, a node with a set of child descendants – which can be either simple tasks or sub-workflows. Each workflow node has a specific functionality in the workflow and has a clearly defined interface, given by its signature, the input and output parameters, and by the exceptions that one of its descendants may *signal* (notify). In addition, in order to *handle* (deal with) the exceptions, each workflow node must have an associated exception handler.

Thus, when enacting a workflow node, its descendants can terminate normally as expected by returning the specified output data or abnormally by signalling an exception. In this case,

the workflow node handler takes control by catching the exception and obtaining a typed data structure which can contain information about the failure context, and then, it accomplishes the required steps for handling the failure.

Currently, the actions a handler can accomplish in DVega are: a) to propagate the exception up in the hierarchy or b) to dynamically replace the signaller (the whole set of descendants of the workflow node) by another set of descendants. In both cases, the enactment of the signaller is aborted before any action is undertaken. With the *propagation* action, a handler can propagate the exception to a level in the workflow hierarchy where it can be processed properly. With the *replacement* action, the handler chooses the most suitable alternative set of child nodes (also referred to as the "alternative candidate") from a list of candidates according to a selection policy, replaces the signaller and starts the enactment of the selected candidate. It is important to note that this replacement is achieved dynamically at run-time. Under specific selection policies, the exception handler may not be able to find a suitable alternative candidate. For these situations, the handler can also propagate the exception up in the hierarchy. There are two main requirements for the signaller to be replaced: (i) both the signaller and the selected candidate can be replaced as long as they have the same signature: the same input and output data. (ii) The workflow node knows nothing about its set of descendants, the node just provides the input data and obtains the output data. In case of exception signalling, the descendants only have to signal it and the workflow node must be aware of handling it. These two characteristics also increase sub-workflow and single task reusability.

There are many different candidate selection policies that may be proposed for the replacement action, ranging from very simple policies such as taking the first candidate of a given candidate list specified at development time [18] to very sophisticated ones such as retrieving the proper candidate from a third party registry of candidates at run-time. Nevertheless, due to the nature of scientific workflows involving long-running scientific tasks and third-party heterogeneous and autonomous resources, we propose three different selection policies:

1. a policy based on selecting the candidate with the least estimated execution time. We propose to estimate the execution time of a candidate by applying the method described in [19], whereby a Petri net performance model is derived automatically from the workflow specification. Then, the model is parameterised and can be fed with different time profiles: constant time values, stochastic variables or even real values obtained by instrumenting the real workflow system. After that, the model can be simulated to obtain a performance prediction. In general terms, accurate and real estimations, however, require prior knowledge of certain workflow aspects related to its performance. For this reason, although the estimations can be computed at run-time, the effectiveness of this policy is constrained by the prior knowledge required.
2. a policy which selects the least expensive candidate. In certain circumstances, third-party resource providers may charge for their service, for instance in terms of a fee per hour of service provided. In those cases, the technique proposed in [19] for predicting the performance of a component can be applied as well. Then, with the component execution

time estimations and knowing the service provider fee, the corresponding component cost predictions can be easily obtained.

3. a policy which selects a candidate considering the context in which the exception was produced. We consider this exception handling policy from an event-condition-action perspective. For instance, consider f_i being a single fault (hardware/software), and $\{f\}$ a set of faults, leading to a *known* event e_i. The event provides certain information about the failure context, so that depending on that context and the conditions specified in the handler, the selection of a candidate is made. Nonetheless, in case no decision could be made (i.e. in case an exception provides a context that was not expected in the handler conditions), the handler can propagate the exception up in the hierarchy.

On the other hand, aborting a signaller in a distributed environment may require the system to undo all the possible, undesired side effects such as stopping running jobs, unreserving resources, delete files, etc. However, all this activities are out of the scope of this paper, some proposals address this issue by following a transactional approach such as in [20]. In our proposal, we deliberately delegate the solving of this problem to the service-oriented infrastructure of DVega which can provide a transactional approach based on Web standards for dealing with these undesired side effects as in [21]. In consequence, DVega's workflow language is unaffected.

Another important aspect is that, in addition to the actions a handler can accomplish that were described here, other actions have also been proposed in the literature [20] such as *retry* or *resume*. The *retry* action can be seen as a particular case of our replacement action, where the signaller is aborted and replaced by the same signaller. The *resume* action does not abort the signaller, but the signaller is provided with some repairing input data which allows the enactment to be continued from the point where the failure arose. This action could be also addressed within our approach and modelled with Reference nets.

4.2. Exception Handling Patterns for Hierarchical Scientific Workflows

In DVega, all workflow node types are implemented by the same pattern, shown in Figure 2. In this section, two patterns for two different actions for exception handling are described. The first one propagates the exception up in the hierarchy and the second one finds an alternative candidate to the signaller, dynamically replaces the signaller with the selected candidate and starts the enactment of the candidate. In case no candidate was found, the pattern propagates the exception up in the hierarchy. Both exception handling patterns extend the initial one of Figure 2 by incorporating an exception handler to it which is expressed by the transition sub-set named `ei` and their related places and arcs.

4.2.1. The exception propagation pattern

The initial pattern of Figure 2 expects the set of child nodes to terminate their execution in the normal way, without any failure. Figure 4 shows an extension of the initial pattern with an exception handler that propagates the exceptions up in the hierarchy. As in the original pattern, the node starts its normal execution in Transition `t1` and ends its normal execution

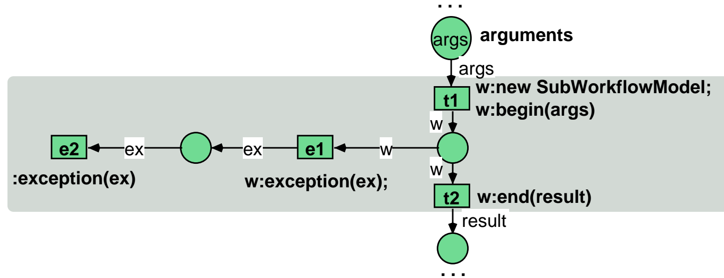

Figure 4. Exception Propagation Pattern

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 be fired in case of normal enactment, whereas Transition `e1` will only be fired in case of exception signalling and the different synchronous channels at both transitions will enable one or another alternatively. Synchronous channel `w:exception(ex)` at Transition `e1` captures the exception from the signaller at a lower level as well as its cause and context (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`.

4.2.2. The exception treatment pattern

Figure 5 shows the pattern that treats an exception by dynamically replacing the signaller. When enacted, at run-time, it receives as an input both the arguments required for the enactment (Variable `args`) and a named list of alternative candidates (Variable `excWf`) for replacing the signaller (either sub-workflows or just simple tasks). As in the exception propagation pattern, there is no indeterminism between Transitions `t2` and `e1`. When an exception is signalled, Transition `e1` is enabled and fired, Variable `ex` will retrieve the exception cause and context, obtained by the synchronous channel. In Transition `e2`, the best candidate of the alternative candidates should be chosen by using one of the predefined policies of Section 4.1: Action `nw=SubWfSelector.select(list,ex)` selects the candidate according to one of the three policies mentioned earlier and assigns a new instance of the selected candidate to Variable `nw`. In case no candidate could have been selected, `nw` is set to null. Transition `e3` can only be enabled and subsequently fired when variable `nw` is not set to null (notice that this is because Transition `t3` has the guard inscription `guard nw!=null`, a constraint that only enables the transition when the condition is true, in this case, `nw` not null). In contrast, when Variable `nw` is set to null the exception will be propagated up in the hierarchy by Synchronous channel `:exception(ex)` in Transition `e4`. After Transition `t3` is fired, the enactment of the selected candidate is started in Transition `t5` (by means of Synchronous channel: `w:begin`).

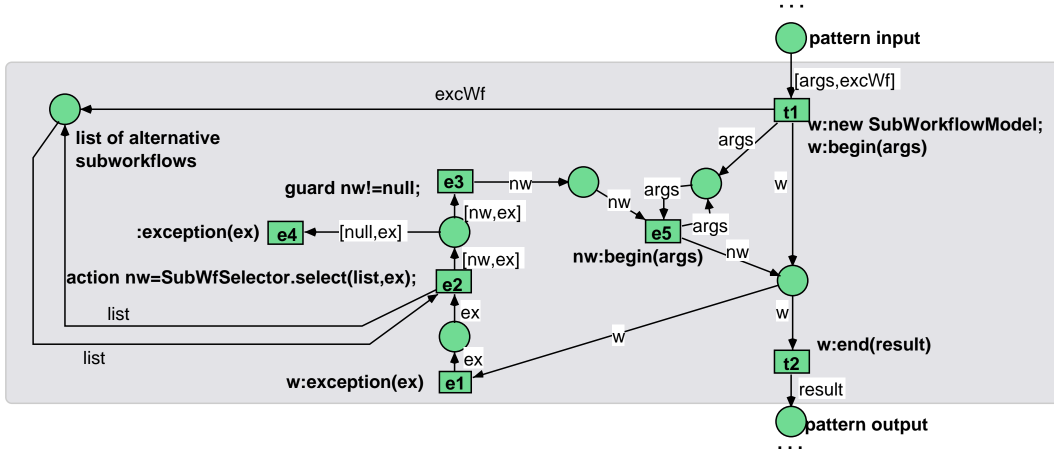

Figure 5. Exception Treatment Pattern

4.3. Implementation: Exception Handling in DVega

Both expressing the exception handling patterns described above with the required dynamics and meeting the requirements exposed in Section 1 can be achieved exclusively in terms of Reference nets without requiring additional mechanisms. In consequence, DVega's workflow interpreter – based on the Renew Reference net interpreter – can enact workflow specifications which incorporate these exception handling mechanisms without any modification. The workflow enactment service, however, requires additional control changes for guaranteeing reliable enactments.

One of these changes is the control during a workflow enactment the system must undertake over the number of consecutive failed executions of a component. Indeed, not only may a component appear in the normal workflow execution flow, but also as an alternative candidate in an exception handler. Therefore, a persistently failing component can lead to serious workflow inefficiency. Besides, depending on the selection policy of the Exception Treatment Pattern, it may even lead to endless loops. For these two reasons, each workflow component in DVega has a maximum failure threshold associated which can be configured at workflow development-time. At run-time, once the threshold is reached, the system shows a pop-up window asking the user to consider a threshold increment or to mark the component as not appropriate for the rest of the enactment.

Additionally, DVega's message broker is another important component in its service-oriented architecture. It allows workflows to communicate with external (middleware) services by exchanging messages and it is also responsible for detecting and raising Middleware and Resource-level exceptions. The bottom part of workflow hierarchies in DVega corresponds to

simple workflow tasks. Thus, in the message broker, once a fault is detected and the exception class is determined, the component tries to gather as much information as possible from the context. Then, the failed simple task in the workflow is signalled the exception which can be either propagated up in the hierarchy or the failed task can be replaced by an alternative task.

5. EXAMPLE

In this section, we use a real and classical example from bioinformatics in order to illustrate our proposal for exception handling. Figure 6 shows a schema of a workflow [‡] from the `myexperiment.org` project which accomplishes a protein sequence analysis: it takes a protein sequence as an input, searches for homologues – other proteins with similar subsequences to the input sequence that may be a consequence of functional, structural or evolutionary relationships – and returns a graphical representation of the homologues (called phylogenetic tree). This process in the workflow of Figure 6 is accomplished in four sequential steps undertaken by four sub-workflows (components): the first one, *Sub-workflow Sequence Similarity Search*, finds the ids of the homologues of a given protein sequence within a database; then, with these ids, the homologues are retrieved by *Sub-workflow Fetch Sequences*; *Sub-workflow Multiple Sequence Alignment* aligns the homologues fetched and finally a graphical representation, a phylogenetic tree, is produced by *Sub-workflow Phylogenetic tree*.

For our illustrating purpose, we are only focused on the first sub-workflow, *Sub-workflow Sequence Similarity Search* and, accordingly, in Figure 6, the details of the rest of the components are deliberately hidden. This sub-workflow performs a database similarity search utilising the BLAST heuristic [22] – for that purpose a set of services of the European Bioinformatics Institute (EMBL-EBI)[§] [23] are involved. One of its tasks, Task `runWUBlast`, creates and starts the BLAST search remotely and then returns the BLAST job identification. Then, there is a sub-workflow inside called `Poll_Job` that monitors the execution status of the job and prevents the subsequent three tasks – `Get_XML_Result`, `Get_Hit_ID_List` and `Get_Text_Result` – from being enacted until the BLAST job terminates correctly. Finally, these three tasks when enacted, will retrieve the results of the database search.

Database similarity searches are fundamental in bioinformatics since they are some of the best ways of gaining information about unknown genes and proteins. Apart from *BLAST*, different methods are available, being both time and accuracy of great significance. Indeed, even though BLAST performs significantly better than other alternatives such as the *Smith-Waterman* [24] algorithm, it provides sub-optimal sets of homologues. However, the *Smith-Waterman* algorithm obtains the optimal set of homologues. In Figure 7, an alternative component to *Sub-workflow Sequence Similarity Search* of Figure 6 is shown. This alternative sub-workflow, also taken from `myexperiment.org`, when properly parameterised, can provide many different search methods, being one of them the *Smith-Waterman* algorithm – in particular, this component also utilises services [23] given by EMBL-EBI. It should be noticed

[‡] http://www.myexperiment.org/workflows/210
[§] http://www.ebi.ac.uk/

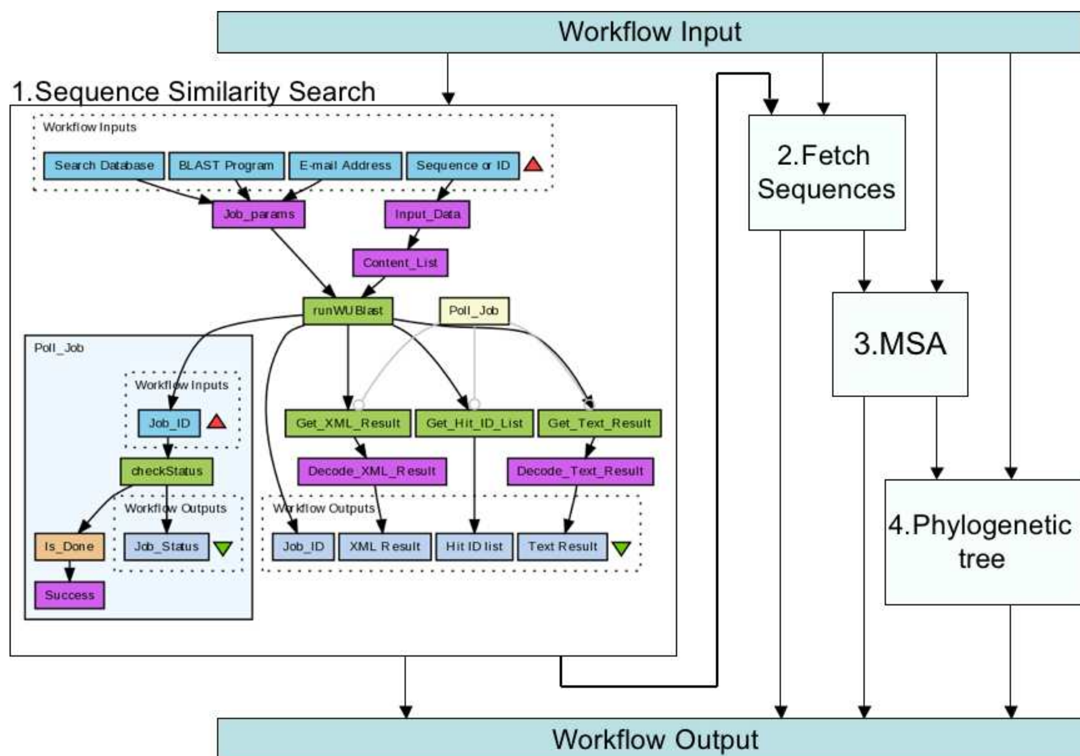

Figure 6. Protein Sequence Analysis workflow taken from `http://www.myexperiment.org/workflows/210`

that both the original sub-workflow and the alternative candidate can be exchanged one another in the original workflow from Figure 6 because their signatures are equivalent both syntactically and semantically.

In order to clarify our exception handling mechanisms better, we propose a scenario in which the Sequence Analysis Workflow from Figure 6 was expressed within DVega's workflow language including the exception handling patterns. However, in spite of the limitations of Petri nets for expressing specific workflow patterns [25] (patterns involving multiple instances, advanced synchronization patterns and cancellation patterns), graph-based languages – like the one used in `myexperiment.org`– do not contain any of these limitations and they can also be expressed with a Reference net [19]. *Sub-workflow Sequence Similarity Search-BLAST* was modelled with an Exception Treatment Pattern and its internal *Sub-workflow Poll_Job* was modelled with an Exception Propagation Pattern.

Figure 8 reproduces the initial steps of the enactment of the workflow. The workflow begins its enactment with *Sub-workflow Sequence Similarity Search-BLAST* which starts a search in

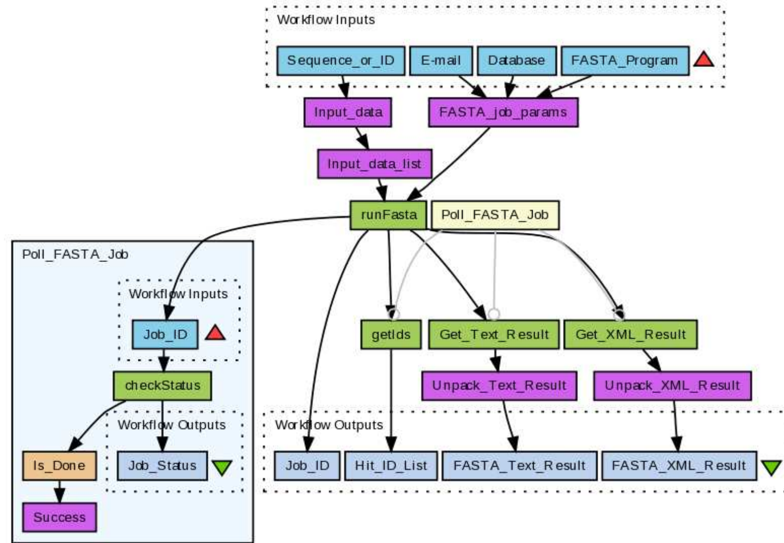

Figure 7. *Sequence Similarity Sub-workflow* based on the Smith-Waterman algorithm alternative to the BLAST heuristic, taken from `http://www.myexperiment.org/workflows/199`

a remote database. Then, *Sub-workflow Poll_Job* checks the status of the search periodically until it is finished. However, let us suppose that this activity fails and an exception is signalled by Task `checkStatus` (notice that it is the child task of *Sub-workflow Poll_Job* responsible for monitoring the status). At this point, *Sub-workflow Poll_Job* stops its execution abnormally and the control, see Figure 8, is given to its Exception Handler. Since, *Sub-workflow Poll_Job* was modelled with an Exception Propagation Pattern, the handler propagates the action to the immediate upper level in the hierarchy. The exception is then caught by the Exception Handler of *Sub-workflow Sequence Similarity Search*. In this case, the pattern chosen was Exception Treatment Pattern, for that reason, a candidate is chosen from the list of candidates. Let us suppose that the selected one is the sub-workflow from Figure 7, a search based on the Smith-Waterman algorithm. The *BLAST* sub-workflow that was being enacted is aborted and then replaced by a new instance of the alternative candidate. As it can be seen in Figure 8, the enactment of the workflow continues at this point with the Smith-Waterman version of the *Sub-workflow Sequence Similarity Search*.

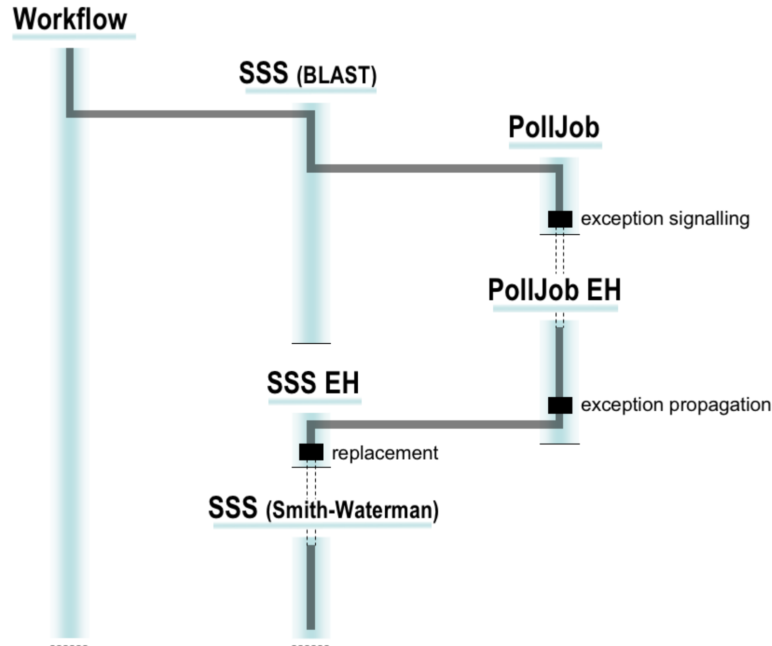

Figure 8. Enactment Diagram of the Sequence Analysis Workflow 210 from `myexperiment.org` with the Exception Handling Mechanisms provided in DVega

6. EVALUATION: scalability

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, e5` and their actions (see Figure 5). This overhead mainly depends on two factors: (i) the selection of an alternative sub-workflow (Transition `e2`), which in this case, for the experiment, it was achieved by taking the first element of a list of names containing candidate sub-workflows: this action can be undertaken in constant time. (ii) The instantiation of the selected alternative sub-workflow (Transition `e2` again) (see Figure 5) which dynamically selects, loads and creates an instance of a sub-workflow. 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 9: showing the sub-workflow 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 ($R^2 = 0.9$) and the minimum overhead obtained is for

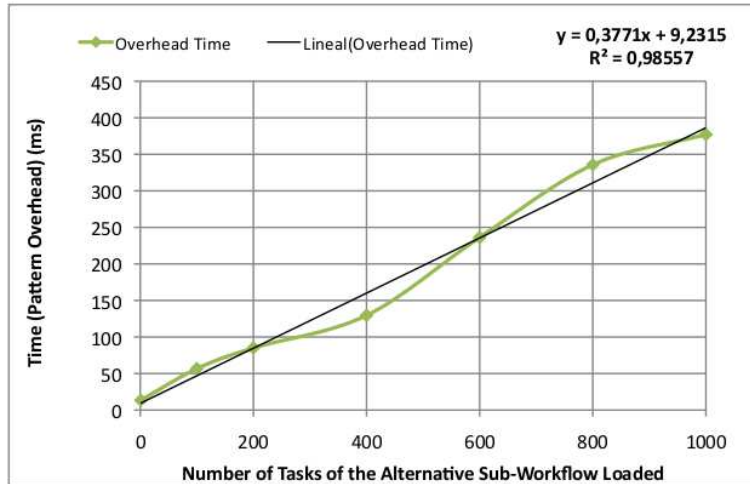

Figure 9. Overhead of the Exception Treatment Pattern

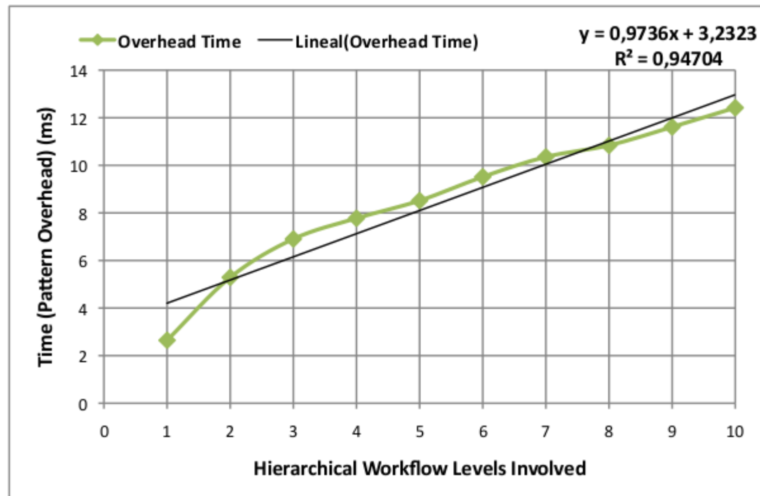

Figure 10. Overhead of the Exception Propagation Pattern

the 1-task sub-workflow (13.48 ms), whereas the maximum overhead (377.14 ms) corresponds to the scenario of the 1000-task sub-workflow.

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 channels. These communication and synchronisation actions are accomplished by Transitions `e1, e2` at each propagation pattern (see Figure 4). 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 10 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 (2.65 ms) and the maximum (12.42 ms) correspond to 1 level of propagation and to 10 levels of propagation, respectively. Therefore, both experiments show that the proposal is scalable. In addition, it is also important to note that, in general terms, the overhead of the proposed patterns is low in comparison to normal execution times of scientific workflows which can be days, weeks or even months.

The experiments were undertaken on a 2.2 GHz Intel Core 2 Duo processor, with 2.5GB 667MHz DDR2 SDRAM and a 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. RELATED WORK

In the business workflow community, a lot of research has been undertaken on exception handling for the last decade. OPERA [20] was one of the first initiatives, based on programming language concepts, which incorporated workflow language primitives for exception handling. OPERA's workflows have a hierarchical nested structure in which nodes are tasks of three different types. Each workflow node has an associated exception handler that can undertake different recovery actions namely *resume*, *retry*, *abort–replace* or *propagate*. Whenever a child task fails, the task signals an exception that is handled by the exception handler of its parent and a recovery action is accomplished. In case the handler undertakes an *abort-replace* action and in order to undo any side effect, the exception handler relies on semantic information provided in the form of compensating tasks and managed using transactional mechanisms. In spite of the many similarities between this approach and our proposal, exception handlers in OPERA do not provide the dynamism that scientific workflows require: handlers are statically defined at development time and do not consider the context failure or any other feature for selecting the appropriate recovery action, limiting thus the flexibility of the approach.

This required flexibility for scientific workflow exception handling is also pursued by [26] in business workflows, using a service–oriented architecture [27]. When an exception occurs, the workflow engine service sends a request for obtaining an *exlet* (an exception handler) to be enacted. The returned *exlet* will be chosen among a group of candidate *exlets* associated with the failed task and the selection will be done by considering the failure context and a set

of rules. The main difference from our approach is that we exclusively express the required flexible exception handling behaviour within the workflow language in terms of Reference nets that can be properly interpreted, instead of achieving the flexibility by means of the infrastructure. The most important advantage of this is that workflow designers can have a better control of the workflow behaviour as it is specified exclusively within the workflow descriptions. Another important difference is that, in [26], in case of an exception requiring compensation tasks for undoing undesired effects, the compensation tasks are incorporated within the *exlet* descriptions explicitly. In our proposal, the compensation tasks does not appear in the exception handler and its accomplishment is relegated to the infrastructure. Therefore, our exception handlers are basically composed of a static part, the required control, and a dynamic part, the component that will be loaded dynamically at run-time. It should be noticed that component re-usability is improved with our approach as components may be part of both the normal executions of the workflow and its exceptional deviations.

Besides, it is also important to highlight that the business workflow language standard, BPEL [29], also provides control structures for exception handling which are based on the exception handling primitives of programming languages. An exception signalled from an activity is propagated up in the scope hierarchy, disabling all other activities in each scope, until a matching handler is found. However, BPEL is not designed to support flexible, dynamic adaptation as required by scientific workflows.

On the other hand, a classification framework for exception handling in business workflows is provided in [30]. First, the range of issues that may lead to exceptions during workflow execution is discussed and an exception type classification for business workflows is given. Then, three main considerations are proposed for determining a set of exception handling patterns: exception handling at workitem level, exception handling at case level and recovery actions. In contrast, our workflow exception handling patterns proposal is based on the actions a handler can accomplish for dealing with a received exception currently *propagation* and *replacement*. Another difference is that our pattern descriptions also show the control issues required for modelling these patterns as they are expressed with the formalism of Reference nets.

In the domain of scientific workflows, fault tolerance mechanisms have also received a lot of attention due to their importance. In this sense, Hwang et al. [3] argued that the generic, heterogeneous and dynamic nature of the Grid requires a new form of failure recovery mechanism which should be able to address these specific requirements. Consequently, they propose a Grid workflow framework with different fault-tolerance techniques for flexible failure handling. The proposed techniques were classified into two different levels, namely task-level and workflow-level. Task-level techniques mask the effects of the execution failure of tasks in the workflow, while workflow-level techniques– being exception handling among them– manipulate the workflow structure. Nonetheless, although the proposed exception handling technique even considers the fault context in the handling for selecting the most suitable candidate at run-time, hierarchical workflow structures are not supported and, therefore, only simple workflow tasks are considered.

A review of the existing Grid workflow systems and its features, including fault tolerance is provided in [15]. Another more recent work [31], reviews the state-of-the-art on fault tolerance mechanisms in Grid workflow systems and highlights that the surveyed workflow systems provide little support at task and user levels for user-defined exceptions.

Due to space limitations, we only review fault tolerance mechanisms – though with special emphasis on techniques related to exception handling – of four popular scientific workflow systems: Taverna, Triana, Kepler and and the Petri-net-based Grid Workflow Execution Service (GWES).

In Taverna [32], 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 sub-workflows 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 sub-workflows cannot be specified nor the fault can be propagated up in the hierarchy.

Triana's [33] 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.

Kepler [34, 35] is another popular workflow system, derived from Ptolemy II and currently under development across a number of scientific data management projects. Kepler features a fault tolerance framework using the data-dependencies recorded by the Kepler Provenance Framework. Fault tolerance is provided with a composite actor called Checkpoint which contains a primary sub-workflow and optionally alternative sub-workflow(s). When a sub-workflow within a Checkpoint produces an error event, all execution within the Checkpoint is stopped. The Checkpoint handles the error itself (by re-executing the primary, or running an alternate sub-workflow), or passes it up the workflow hierarchy. The maximum number of times to retry the primary or an alternative sub-workflow is configurable. Once the retry limit is exceeded, the error is sent up the workflow hierarchy to the nearest enclosing Checkpoint. Nonetheless, there is no indication in these references regarding the policies undertaken for selecting a candidate from the alternative candidates: no evidence is given whether the context in which the exception arose can be taken into account for selecting the candidate. Besides, the candidate list has to be defined at development time, thus limiting the flexibility and dynamism of the approach.

The Grid Workflow Execution Service (GWES) [2] is a workflow system for supporting hierarchical workflows and handling exceptions with the flexibility and dynamism required by scientific workflows. The GWES processes workflows that are described using the Grid Workflow Description Language (GWorkflowDL), which is based on the formalism of High-Level Petri nets (HLPNs). In addition, in accordance with the Petri net refinement paradigm, places and transitions within the net can be refined with additional Petri nets, thereby facilitating the modeling of large realworld systems. However, the GWorkflowDL does not support the inherent modeling of the dynamic refinement process itself. For instance, there is no simple construct in the GWorkflowDL 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. Complex exception handling needs to be modeled explicitly as part of the application workflow. The GWES does not support a clear separation of the exception handling from the application data and control flow, apart from simple rescheduling techniques and checkpoint/restart functionalities. Our proposal, in contrast, is based on Reference nets, which provide mechanisms inside the formalism for modelling all of these issues thereby users can have a better control of the workflow behaviour.

Finally, there is a group of proposals that, once the excepction is detected and redirected to the corresponding event handler, they provide support at the decision taking system component that should decide the action to perform. Since in large complex systems, this decision is not simple, a frequent approach has been the use of Event Condition Action (ECA) rules to support conten dependent reasoning [36]. The use of ECA-rules has been also considered for adaptative exception handling. These approaches use declarative ECA rules to reason and model the handling alternative strategies instead of employ the same notation as that used for representing workflow processes [37]. In the pattern presented in this work, different decision support system strategies may be used depending on the context failure. Anyway, once a decision has been taken, the alternative is expressed in the same formalism as the workflows process. In this way, declarative and imperative formalism are well-suited in the workflow process modelling.

8. CONCLUSIONS

Inspired by the exception handling mechanisms developed for programming languages, we propose an exception handling mechanism for dynamically adapting the workflow structure of hierarchical scientific workflows at run-time. In contrast to other previous proposals which achieve the required flexibility by means of the infrastructure, our proposal expresses the exception handling mechanisms within the workflow language in terms of two exception handling patterns which are exclusively based on the Reference Nets-within-Nets formalism (a specific type of Petri nets). Therefore, workflow users can have a better control and understanding of the behaviour of their workflows without having to be aware of the underlying infrastructure. Although we appreciate that a Petri net-based formalism is more complex (in terms of representation) than a task graph representation, we believe that the added benefits of supporting exception handling and dynamic creation of sub-workflows overcomes these limitations.

In our proposal, each workflow node in the hierarchy has an associated exception handler. The actions a handler can accomplish are: a) to propagate the exception up in the hierarchy to a level within the workflow hierarchy where it can be processed properly or b) to dynamically replace a workflow node signalling (notifying) an exception by another alternative workflow node. With the *replacement* action, the handler chooses the most suitable alternative workflow node from a list of candidates according to a selection policy and dynamically replaces the signaller, allowing the continuation of the workflow enactment. Thus, in addition to making a more fault-tolerant system, this also increases both sub-workflow and simple task re-usability. For the selection process, we propose three different policies: Event-Condition-Action (ECA) rules considering the failure context, a policy based on choosing the candidate with the least

execution time estimation and a policy based on choosing the candidate with the lowest cost estimation. These two exception handling actions were designed by two Reference net-based exception handling patterns which were integrated and implemented in the DVega workflow system: (i) an exception treatment pattern; and (ii) an exception propagation pattern.

Finally, we use our approach for a real hierarchical workflow example taken from the `myexperiment.org` project and provide experimental results for the scalability analysis of the approach. Our focus in doing this is to demonstrate 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.

REFERENCES

1. Gil Y, Deelman E, Ellisman M, Fahringer T, Fox G, Gannon D, Goble C, Livny M, Moreau L, Myers J. Examining the challenges of scientific workflows. *IEEE Computer* 2007; :26–34.
2. Hoheisel A. User tools and languages for graph-based grid workflows: Research articles. *Concurr. Comput. : Pract. Exper.* 2006; **18**(10):1101–1113, doi:http://dx.doi.org/10.1002/cpe.v18:10.
3. Hwang S, Kesselman C. Grid workflow: A flexible failure handling framework for the grid. *Proc. of the 12th IEEE International Symposium on High Performance Distributed Computing*, 2003.
4. Tolosana-Calasanz R, Bañares JA, Álvarez P, Ezpeleta J. Vega: a service-oriented grid workflow management system. *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS*, *LNCS*, vol. 4804, Springer, 2007.
5. Murata T. Petri nets: Properties, analysis and applications. *Proceedings of IEEE*, vol. 77, 1989; 541–580.
6. vanderAalst W, vanHee K. *Workflow Management: Models, Methods, and Systems.* MIT Press: Cambridge, MA, USA, 2004.
7. Vossberg M, Hoheisel A, Tolxdorff T, Krefting D. A reliable dicom transfer grid service based on petri net workflows. *CCGRID '08: Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, IEEE Computer Society: Washington, DC, USA, 2008; 441–448, doi:http://dx.doi.org/10.1109/CCGRID.2008.122.
8. Pellegrini S, Hoheisel A, Giacomini F, Ghiselli A. Using gworkflowdl for middleware-independent modeling and enactment of workflows. *Proceedings of the CoreGRID Integration Workshop 2008*, 2008.
9. Guan Z, Hern F, Bangalore P, Gray J, Skjellum A, Velusamy V, Liu Y. Grid-flow: A grid-enabled scientific workflow system with a petri net-based interface. *Technical Report*, Exper 2004.
10. Aversano L, Cimitile A, Gallucci P, Villani ML. Flowmanager: A workflow management system based on petri nets. *COMPSAC '02: Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment*, IEEE Computer Society: Washington, DC, USA, 2002; 1054–1059.
11. Valk R. Petri nets as token objects - an introduction to elementary object nets. *LNCS: 19th Int. Conf. on Application and Theory of Petri Nets, Lisbon, Portugal* Jun 1998; **1420**:1–25.
12. Kummer O, Wienberg F, Duvigneau M, Schumacher J, Köhler M, Moldt D, Rölke H, Valk R. An extensible editor and simulation engine for petri nets: Renew. *Applications and Theory of Petri Nets 2004, 25th International Conference, ICATPN'04* 2004; **3099**:484–493.
13. Álvarez P, Bañares JA, Ezpeleta J. Approaching web service coordination and composition by means of petri nets. the case of the nets-within-nets paradigm. *ICSOC*, *Lecture Notes in Computer Science*, vol. 3826, Benatallah B, Casati F, Traverso P (eds.), Springer, 2005; 185–197.
14. Russell N, ter Hofstede A, van der Aalst W, Mulyar N. Workflow control-flow patterns: A revised view. *Technical Report*, BPM Center Report BPM-06 22, BPMcenter.org 2006.
15. Yu, Jia and Buyya, Rajkumar . A Taxonomy of Workflow Management Systems for Grid Computing. *CoRR* 2005; **abs/cs/0503025**.
16. Goodenough JB. Exception handling: issues and a proposed notation. *Commun. ACM* 1975; **18**(12):683–696, doi:http://doi.acm.org/10.1145/361227.361230.
17. Yemini S, Berry DM. A modular verifiable exception handling mechanism. *ACM Trans. Program. Lang. Syst.* 1985; **7**(2):214–243, doi:http://doi.acm.org/10.1145/3318.3320.

18. Tolosana-Calasanz R, Rana O, Rana OF, nares JAB, Álvarez P, Ezpeleta J. Exception handling patterns for hierarchical scientific workflows. *Proceedings of the 6th International Workshop on Middleware for Grid Computing*, ACM Digital Library, 2008.
19. Tolosana-Calasanz R, Rana OF, Bañares JA. Automating performance analysis from taverna workflows. *CBSE*, *Lecture Notes in Computer Science*, vol. 5282, Chaudron MRV, Szyperski CA, Reussner R (eds.), Springer, 2008; 1–15.
20. Hagen C, Alonso G. Exception handling in workflow management systems. *IEEE Trans. Softw. Eng.* 2000; **26**(10):943–958, doi:http://dx.doi.org/10.1109/32.879818.
21. Fabra J, Álvarez P, Bañares JA, Ezpeleta J. A framework for the development and execution of horizontal protocols in open bpm systems. *Business Process Management*, *Lecture Notes in Computer Science*, vol. 4102, Dustdar S, Fiadeiro JL, Sheth AP (eds.), Springer, 2006; 209–224.
22. Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ. Basic local alignment search tool. *J Mol Biol* October 1990; **215**(3):403–410, doi:http://dx.doi.org/10.1006/jmbi.1990.9999. URL `http://dx.doi.org/10.1006/jmbi.1990.9999`.
23. Labarga A, Valentin F, Anderson M, Lopez R. Web services at the european bioinformatics institute. *Nucl. Acids Res.* 2007; **35**:1–6.
24. Smith TF, Waterman MS. Identification of common molecular subsequences. *Journal of Molecular Biology* March 1981; **147**(1):195–197, doi:http://dx.doi.org/10.1016/0022-2836(81)90087-5. URL `http://dx.doi.org/10.1016/0022-2836(81)90087-5`.
25. van der Aalst WMP, ter Hofstede AHM. Workflow patterns: On the expressive power of (petri-net-based) workflow languages. *Proc. of the Fourth International Workshop on Practical Use of Coloured Petri Nets and the CPN Tools, Aarhus, Denmark*, Technical Report DAIMI PB-560, 2002; 1–20.
26. Adams M, ter Hofstede AHM, van der Aalst WMP, Edmond D. Dynamic, extensible and context-aware exception handling for workflows. *OTM Conferences (1)*, Lecture Notes in Computer Science, Springer-Verlag, 2007; 95–112.
27. Adams M, ter Hofstede AHM, Edmond D, van der Aalst WMP. Worklets: A service-oriented implementation of dynamic flexibility in workflows. *OTM Conferences (1)*, Lecture Notes in Computer Science, Springer-Verlag, 2006; 291–308.
28. Hagen C, Alonso G. Flexible exception handling in the opera process support system. *ICDCS*, 1998; 526–533.
29. Curbera F, Khalaf R, Nagy W, Weerawarana S. Implementing bpel4ws: the architecture of a bpel4ws implementation. *Concurrency and Computation: Practice and Experience* 2006; **18**(10):1219–1228.
30. Russell N, van der Aalst WMP, ter Hofstede AHM. Workflow exception patterns. *CAiSE*, Lecture Notes in Computer Science, Springer-Verlag, 2006; 288–302.
31. Plankensteiner K, Prodan R, Fahringer T, Kertesz A, Kacsuk PK. Fault-tolerant behaviour in state-of-the-art grid workflow management systems. *Technical Report*, CoreGRID 2007.
32. Oinn T, Greenwood M, Addis M, Alpdemir MN, Ferris J, Glover K, Goble C, Goderis A, Hull D, Marvin D, *et al.*. Taverna: lessons in creating a workflow environment for the life sciences: Research articles. *Concurr. Comput. : Pract. Exper.* 2006; **18**(10):1067–1100, doi:http://dx.doi.org/10.1002/cpe.v18:10.
33. Churches D, Gombas G, Harrison A, Maassen J, Robinson C, Shields M, Taylor I, Wang I. Programming scientific and distributed workflow with triana services: Research articles. *Concurr. Comput. : Pract. Exper.* 2006; **18**(10):1021–1037, doi:http://dx.doi.org/10.1002/cpe.v18:10.
34. Bowers S, Ludascher B, Ngu AHH, Critchlow T. Enabling scientificworkflow reuse through structured composition of dataflow and control-flow. *ICDEW '06: Proceedings of the 22nd International Conference on Data Engineering Workshops*, IEEE Computer Society: Washington, DC, USA, 2006; 70.
35. Ngu A, Bowers S, Haasch N, McPhillips T, Critchlow T. Flexible scientific workflow modeling using frames, templates, and dynamic embedding. *Scientific and Statistical Database Management* 2008; :566–572.
36. Luo Z, Sheth A, Kochut K, Miller J. Exception handling in workflow systems. *Applied Intelligence* 2000; **13**(2):125–147, doi:http://dx.doi.org/10.1023/A:1008388412284.
37. Mallya AU, Singh MP. Modeling exceptions via commitment protocols. *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, ACM: New York, NY, USA, 2005; 122–129, doi:http://doi.acm.org/10.1145/1082473.1082492.