2012 IEEE Ninth International Conference on Services Computing

A User-Defined Exception Handling Framework in the VIEW Scientific Workflow Management System

Dong Ruan*, Shiyong Lu*, Aravind Mohan*, Xubo Fei*, Jia Zhang†
*Scientific Workflow Research Laboratory, Wayne State University, Detroit, MI, USA
{druan, shiyong, amohan, xubo}@wayne.edu
†Department of Computer Science, Northern Illinois University, DeKalb, IL, USA
{jiazhang}@cs.niu.edu

***Abstract*—With the advances of e-Science, scientific workflow has become an important tool for researchers to explore scientific discoveries. Although several scientific workflow management systems (SWFMSs) have been developed, their support of exception handling is still limited. In this paper, we introduce our approach of exception handling in the VIEW scientific workflow management system. We propose an exception handling language for scientific workflows based on our workflow model. Both syntax and semantics rules of our language are presented. Different exception handling primitives, such as retry, alternative, and repeat, are supported in our language with flexibility for their composition to provide a sophisticated and flexible exception handling mechanism. Moreover, two exception handling algorithms and the architecture design for exception handling in VIEW are also presented.**

***Keywords*-scientific workflow; exception handling; VIEW.**

I. INTRODUCTION

With the advances of e-Science, scientific workflow has become a popular and important tool for researchers and scientists to explore scientific discoveries and accelerate the cycle of scientific experiments. However, the complexity of workflow design and heterogeneous architecture of computing resources may lead to many exceptions during workflow execution. Those exceptions often interrupt normal execution of workflows and result in incorrect results. Moreover, scientific workflows are dataflow-oriented and often have long time duration. Exception handling can save scientists' time of restarting the whole long-running workflows and help scientists better carry on scientific experiments. Therefore, exception handling plays an important role in the context of scientific workflows.

Based on our literature review of exception handling [1] in scientific workflows, we have the following observations:

- Sophisticated and flexible exception handling. Scientific workflow is a formalization of a scientific computation process, which often involves complicated, heterogeneous tasks, and distributed computation resources. Single exception handling technique is not sufficient for this challenge, more compound approach is needed. On the other hand, SWFMSs should be friendly enough for scientists to perform different exception handling techniques. Therefore, not only sophisticated, but also flexible exception handling mechanisms are needed. Challenge lies on the balance of sophistication and flexibility.
- Exception handling languages in scientific workflow. Although much research has been done on scientific workflow, there are no generic models or frameworks specifically for exception handling in scientific workflows yet. Such a generic exception handling language for scientific workflow can provide a formal foundation for workflow modeling and open the way to consider exception handling in scientific workflow from a more general point of view since language is syntactically abstracted and multi-paradigm friendly.

The remainder of this paper is organized as follows, section II proposes a scientific workflow exception handling language in VIEW [2] [3], including the syntax of our language as well as illustration examples. Section III presents the semantics rules of our language. Section IV introduces two exception handling algorithms, EHParse and HandleException. Section V introduces exception handling architecture design in VIEW, section VI discusses related work and section VII gives the conclusions and future work.

II. A SCIENTIFIC WORKFLOW EXCEPTION HANDLING LANGUAGE IN VIEW

In scientific workflows, tasks (aka. actors, processors, etc.) were considered as the atomic building components in a workflow. Recently, in [4], a scientific workflow model for VIEW was introduced. In the proposed model, workflows were considered as atomic building components. Primitive workflows were used to represent traditional tasks as the atomic building blocks, graph based workflows were used to represent workflows that consist of multiple primitive workflows. This model elegantly solves the workflow composition issue by having workflows as the only operands. However, this model has few exception handling features. In order to better support exception handling, we introduce our exception handling language in the VIEW scientific workflow management system. Our proposed language is based on our workflow model. First of all, we present the syntax of our language, and then we show some examples of our language.

We define the abstract syntax (in Backus Normal Form) of our exception handling language as follows:

P ::= *retry(k)* | *alternative(t)* | *delay(k)* | *seq*(P_1, P_2) | *repeat(P, n)*

- *P*, P_1,P_2 are exception handlers.
- Symbol "::=" should be read as "is defined as" and symbol "|" as "or".
- *retry(k)* represents to rerun current workflow k times, until either exception is handled successfully or k times are used up, whichever comes first.
- *alternative(t)* represents to call another equivalent workflow t.
- *delay(k)* represents to pause the execution of workflow for k milliseconds.
- *seq*(P_1, P_2) represents the sequential execution of two exception handlers, P_1 and P_2.
- *repeat(P, n)* represents to repeat the execution of an exception handler *P* for *n* times, until either exception is handled successfully or n times are used up, whichever comes first.

Tranditional exception handling techniques, such as retry, alternative, are supported in our language. Common control structures, such as sequential, loop, are also covered. This gives us the power to build sophisticated exception handlers with hierarchical structures. End users can use our language according to different computing resources and the nature of certain workflow tasks. For example, if a web service we are using in our workflow is deployed on a unstable resource, it might be helpful to use retry since the resource is not stable, or an stabler alternative one can also help. Furthermore, *seq* and *repeat* can add the ability to build sophisticated exception handlers and provide control for the execution of different parts in the exception handlers. Given the abstract syntax proposed above, we can build different kinds of exception handlers with sophistication and flexibility. For example, P_1 = *retry(1)* means to rerun the current workflow once; P_2 = *seq(delay(100), alternative(105))* means to pause the current execution for 100 milliseconds first, then invoke an alternative workflow 105. P_3 = *seq(alternative(101),* P_2*)* means to invoke workflow 101 first, if the exception is not handled successfully, invoke exception handler P_2; P_4 = *seq*(P_1,P_1) means to sequentially try exception handler P_1 twice, if the first try does not handle the exception successfully, the second try will be invoked, if the first try succeeds in handling the exception, the second try will not be invoked; similarly, P_5 = *repeat*(P_3, *3*) means to repeatedly execute exception handler P_3 for 3 times until the exception is handled in any of the three tries or 3 times is used up, whichever comes first.

In VIEW, exception handlers are embedded as part of the workflow specification. One VIEW workflow example is given in Figure 1. *MathWorkflow2* has four inputs and one output, it contains one primitive workflow: *AddTwoDouble* and one nested workflow, *addsubWSnested*. *AddTwoDouble* adds two inputs of *MathWorkflow2*, the sum and other two inputs are used by nested workflow: *addsubWSnested*. *addsubWSnested* has two primitive workflows: *AddTwoNumber* and *SubTwoNumber*. The output of *addsubWSnested* is the output of *MathWorkflow2*. In *MathWorkflow2*, primitive workflows are all implemented using web services.

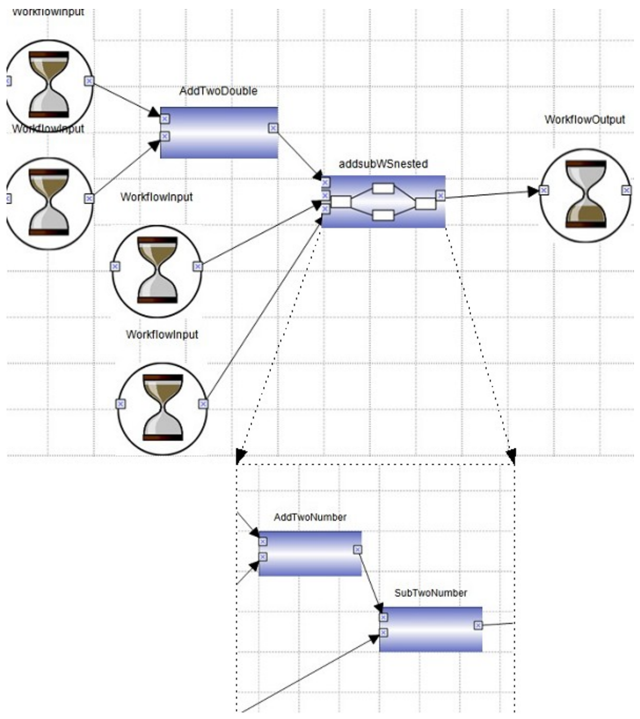
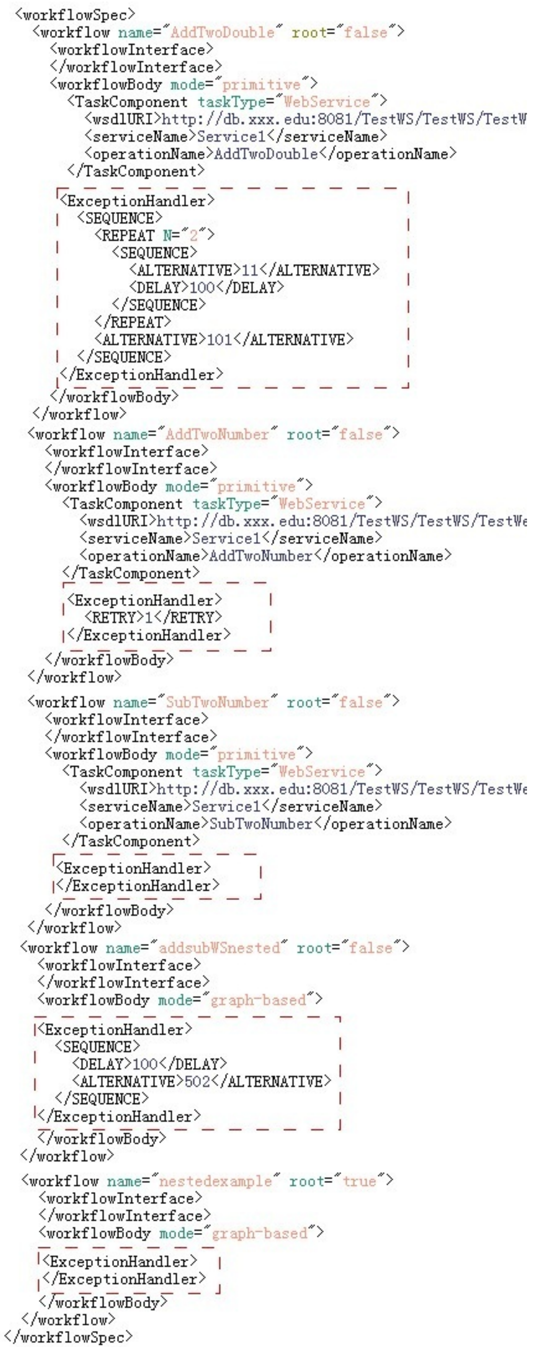


Figure 1. *MathWorkflow2* (composite structure).

Figure 2 give the corresponding workflow specification (in XML) of *MathWorkflow2*. In our workflow specification, multiple workflows are included. Moreover, different layers are separated using the following elements: the *workflowInterface* element defines the workflow input and output information, the *workflowBody* element defines the functional parts of workflows, and the *ExceptionHandler* element defines the exception handlers for the workflows. For one workflow, its exception handlers are embedded within workflow body in the specification. The *root* element defines the entry point among the set of workflows for the workflow engine to execute a workflow. Exception handlers are built in the specification files following the syntax of our language. We illustrate the given example by representing exception handlers in example specification file using our language statement. In Figure 2, there are five exception handlers. The exception handler for *AddTwoDouble*, $P_{AddTwoDouble}$ = *seq(repeat(seq(alternative(11), delay(100)), 2), alternative(101))*, tells the engine to repeat workflow 11 with an interval of 100 miliseconds 2 times, if the exception is not handled, then try workflow 101. The exception handler for *AddTwoNumber*, $P_{AddTwoNumber}$ = *retry(1)*, means to retry the same workflow once. The exception handler for *SubTwoNumber*, $P_{SubTwoNumber}$, is empty. The exception handler for nested workflow, *addsubWSnested*, $P_{addsubWSnested}$ = *seq(delay(100), alternative(502))*, represents to pause for 100 miliseconds then invoke workflow 502. The exception handler for *nestedexample*, $P_{nestedexample}$ is empty.

Using our proposed language, we can build exception handlers with different levels of sophistication. Exception handlers can have nested structures or be empty. These exception handlers cover most execution patterns in scientific workflow, such as sequential, loop. Moreover, different exception handling techniques, such as retry, alternative, delay, are embedded in our exception handlers. On the other hand, our language provides relatively simple format and easy composition for users, integrates flexibility of bundling different execution patterns together to build exception handlers. Our language lies on the balance point of sophistication and flexibility of scientific workflow exception handling. The examples above illustrate the power of our proposed language:

- Sophisticated exception handlers. Our proposed exception handling language provides the power to build sophisticated exception handlers for scientific workflows. Sophisticated exception handlers can have nested structure, different control structures, as well as common exception techniques embedded. Those powerful exception handlers can do a better job to ensure the successful execution of scientific workflows.
- Flexible exception handlers. Our proposed exception handling language elegantly wraps different control structures and traditional exception handling techniques using a relatively simple syntax. Exception handlers can be built flexibly using easy grammar without deep knowledge of scientific workflows.
- Hierarchical exception handlers. Our proposed exception handling language can provide exception handlers for both simple workflows and composite workflows. For composite workflows, our language can specify exception handlers for both subworkflows inside the nested structure and the composite workflow itself, therefore, our language supports exception handling for hierarchical workflow structures.
- Modular exception handlers. Using our proposed exception handling language can build modular exception handlers, sophisticated exception handlers can be formed using simpler exception handlers. Basic exception handlers can be reused in different exception handlers. So our language support the modular design and reusability of exception handlers.



```
<workflowSpec>
  <workflow name="AddTwoDouble" root="false">
    <workflowInterface>
    </workflowInterface>
    <workflowBody mode="primitive">
      <TaskComponent taskType="WebService">
        <wsdlURI>http://db.xxx.edu:8081/TestWS/TestWS/TestW
        <serviceName>Service1</serviceName>
        <operationName>AddTwoDouble</operationName>
      </TaskComponent>
     <ExceptionHandler>
       <SEQUENCE>
         <REPEAT N="2">
           <SEQUENCE>
             <ALTERNATIVE>11</ALTERNATIVE>
             <DELAY>100</DELAY>
           </SEQUENCE>
         </REPEAT>
         <ALTERNATIVE>101</ALTERNATIVE>
       </SEQUENCE>
     </ExceptionHandler>
    </workflowBody>
  </workflow>
  <workflow name="AddTwoNumber" root="false">
    <workflowInterface>
    </workflowInterface>
    <workflowBody mode="primitive">
      <TaskComponent taskType="WebService">
        <wsdlURI>http://db.xxx.edu:8081/TestWS/TestWS/TestWe
        <serviceName>Service1</serviceName>
        <operationName>AddTwoNumber</operationName>
      </TaskComponent>
      <ExceptionHandler>
        <RETRY>1</RETRY>
      </ExceptionHandler>
    </workflowBody>
  </workflow>
  <workflow name="SubTwoNumber" root="false">
    <workflowInterface>
    </workflowInterface>
    <workflowBody mode="primitive">
      <TaskComponent taskType="WebService">
        <wsdlURI>http://db.xxx.edu:8081/TestWS/TestWS/TestWe
        <serviceName>Service1</serviceName>
        <operationName>SubTwoNumber</operationName>
      </TaskComponent>
     <ExceptionHandler>
     </ExceptionHandler>
    </workflowBody>
  </workflow>
 <workflow name="addsubWSnested" root="false">
   <workflowInterface>
   </workflowInterface>
   <workflowBody mode="graph-based">
   <ExceptionHandler>
     <SEQUENCE>
       <DELAY>100</DELAY>
       <ALTERNATIVE>502</ALTERNATIVE>
     </SEQUENCE>
   </ExceptionHandler>
   </workflowBody>
 </workflow>
 <workflow name="nestedexample" root="true">
   <workflowInterface>
   </workflowInterface>
   <workflowBody mode="graph-based">
    <ExceptionHandler>
    </ExceptionHandler>
   </workflowBody>
 </workflow>
</workflowSpec>
```

Figure 2. Specification of *MathWorkflow2*.

III. Semantics rules

In this section ,we present the operational semantics rules of our proposed exception handling language. These semantics rules are defined in terms of a set of inference rules in a structural way, all the rules can be composed together to interpret execution of complex exception handlers. We introduce some notations we use in our inference rules, as well as some assumptions as follows:

- *w* : we use *w* to represent a workflow that can be either primitive or composite.
- *P* : we use *P* to represent an exception handler, ϕ is used to represent an empty exception handler.
- $\overrightarrow{i}$: we use $\overrightarrow{i}$ to represent the inputs of a workflow. One workflow can have zero, one, or more than one input.
- *s* : we use *s* to represent the states of a workflow execution. *s* ::= *T* | *F*, means s can be either *T* or *F*. *T* stands for True, meaning no exceptions occur; *F* stands for False, meaning exceptions happen.
- Function $f : \overrightarrow{i} \times w \times s \rightarrow s$ is used to model the semantics of execution of a particular workflow in one exception handler. Hence, after workflow execution, the resulting state will be *f*($\overrightarrow{i}$, *w, s*).
- Triple ($\overrightarrow{i}$, *w*, *P*) is used to represent the invocation of an exception handler *P* of workflow *w* with input $\overrightarrow{i}$ and *(s,(*$\overrightarrow{i}$*, w, P))* is used to represent to invoke exception handler *P* of workflow *w* with input $\overrightarrow{i}$ at state *s*.

The inference rules of our exception handling language are formalized in Figure 3. For each syntax rule, we present inference rules to formalize its semantic meaning. We explain each rule in the following:

- The delay rule describes the semantics of executing a *delay(k)* exception handler. Intuitively, it states that "pause the current workflow execution for *k* milliseconds". Essentially, *delay(k)* does not change the state of execution. Two rules are introduced: rule 1.1 reduces the pause time by 1 millisecond and rule 1.2 covers the situation of delay time is zero. This rule is an axiom since it has no hypothesis. Similar interpretations hold for other rules.
- The alternative rule describes the semantics of executing exception handler, *alternative(t)*. Intuitively, it states that "invoke workflow *t* as an alternative". Two rules are introduced: rule 2.1 says that if the execution of workflow *t* is helpful, meaning the exception is handled, then state is changed from *F* to *T* and exception handler becomes empty; rule 2.2 says that if after the execution of workflow *t*, the exception is not handled, the state is still *F*, and exception handler becomes empty.
- The retry rule describes the semantics of executing exception handler, *retry(k)*. Intuitively, it states that "rerun current workflow *k* times, and stop until either exception is handled or *k* times are used up, whichever comes first". Three rules are presented: rule 3.1 says if the first time retry handles exceptions, then state changes to *T*, and exception handler becomes empty; rule 3.2 says that if the first time retry does not handle the exception, then another try will be taken, *retry(k)* reduces to *retry(k-1)*; rule 3.3 explains the situation when *k* is zero, which does not change the state.

1. delay rule:

$$\frac{}{(F,(\vec{i},w,delay(k)))\rightarrow(F,(\vec{i},w,delay(k-1)))}\ (k>=1) \qquad (1.1)$$

$$\frac{}{(F,(\vec{i},w,delay(0)))\rightarrow(F,(\vec{i},w,\phi))}\ (k=0) \qquad (1.2)$$

2. alternative rule:

$$\frac{f(\vec{i},t,F)=T}{(F,(\vec{i},w,alternative(t)))\rightarrow(T,(\vec{i},w,\phi))} \qquad (2.1)$$

$$\frac{f(\vec{i},t,F)=F}{(F,(\vec{i},w,alternative(t)))\rightarrow(F,(\vec{i},w,\phi))} \qquad (2.2)$$

3. retry rule:

$$\frac{f(\vec{i},w,F)=T}{(F,(\vec{i},w,retry(k)))\rightarrow(T,(\vec{i},w,\phi))}\ (k>=1) \qquad (3.1)$$

$$\frac{f(\vec{i},w,F)=F}{(F,(\vec{i},w,retry(k)))->(F,(\vec{i},w,retry(k-1)))}\ (k>=1) \qquad (3.2)$$

$$\frac{}{(F,(\vec{i},w,retry(0)))->(F,(\vec{i},w,\phi))}\ (k=0) \qquad (3.3)$$

4. seq rule:

$$\frac{(F,(\vec{i},w,P_1))\rightarrow(T,(\vec{i},w,\phi))}{(F,(\vec{i},w,seq(P_1,P_2)))\rightarrow(T,(\vec{i},w,\phi))} \qquad (4.1)$$

$$\frac{(F,(\vec{i},w,P_1))\rightarrow(F,(\vec{i},w,\phi))}{(F,(\vec{i},w,seq(P_1,P_2)))\rightarrow(F,(\vec{i},w,P_2))} \qquad (4.2)$$

5. repeat rule:

$$\frac{(F,(\vec{i},w,P))\rightarrow(T,(\vec{i},w,\phi))}{(F,(\vec{i},w,repeat(P,n)))\rightarrow(T,(\vec{i},w,\phi))}\ (n>=1) \qquad (5.1)$$

$$\frac{(F,(\vec{i},w,P))\rightarrow(F,(\vec{i},w,\phi))}{(F,(\vec{i},w,repeat(P,n)))\rightarrow(F,(\vec{i},w,repeat(P,n-1)))}\ (n>=1) \qquad (5.2)$$

$$\frac{}{(F,(\vec{i},w,repeat(P,0)))\rightarrow(F,(\vec{i},w,\phi))}\ (n=0) \qquad (5.3)$$

6.

$$\frac{}{f(\vec{i},w,F)\equiv f(\vec{i},w-P(w),F)} \qquad (6.1)$$

Figure 3. Semantics rules of our language.

- The seq rule describes the semantics of executing exception handler, *seq*(P_1,P_2). Intuitively, it states that "sequentially try exception handlers: P_1 and P_2". Two rules are introduced: rule 4.1 says if P_1 handles the exception, then the execution of *seq*(P_1,P_2) is done, state changes from *F* to *T*; rule 4.2 says if P_1 does not handle the exception, P_2 will be executed.
- The repeat rule describes the semantics of executing exception handler, *repeat(P,n)*. Intuitively, it states that "repeatedly execute exception handler *P n* times, and stop until either exception is handled or *n* times are used up, whichever comes first". Three rules are introduced: rule 5.1 says if the first execution of *P* handles the exception, then state changes to *T*, execution is done; rule 5.2 says if the first execution of *P* does not handle

the exception, another attempt will be made, and the total number of attempts will decrease one; rule 5.3 explains the situation when *n* is zero, which does not change the state.
- Rule 6.1 is an axiom. This rule tells that if a workflow, *w*, is used in one exception handler, *P*, and *w* itself has an exception handler, *P(w)*, in its body. Then *P(w)* will not be invoked during the execution of *P*. "$\equiv$" means execution of the left side and the execution of right side are semantically equivalent.

We demonstrate the execution of our example exception handler using the inference rules we defined. Given $P_{AddTwoDouble}$ = *seq(repeat(seq(alternative(11), delay(100)), 2), alternative(101))*, according to the meaning of each pattern, the execution of $P_{AddTwoDouble}$ will start from *alternative(11)*, alternative rule will be applied. Let P_1 = *seq(alternative(11), delay(100))*, P_2 = *repeat(P_1,2)*, $P_{AddTwoDouble}$ =*seq(P_2, alternative(101))*. One successful execution of $P_{AddTwoDouble}$ is given below in Figure 4 using inference rules in Figure 3. From line 1 to line 5, it reads as hypothesis and conclusion. Line 1 is a hypothesis, which means the execution of workflow 11 successfully handles exceptions, rule 2.1 applies, and we get conclusion in line 2, which says exception is handled, state is changed from F to T and exception handler becomes empty. Similarly, line 2 can be the hypothesis of line 3, by applying rule 4.1, which tells for a *seq* pattern, *alternative(11)* handles the exception, then the execution of *seq(alternative(11), delay(100))* is done, state changes to *T* and exception handler becomes empty. Then applying rule 5.1, using line 3 as hypothesis, we get the conclusion in line 4. This says that the first execution of P_1 = *seq(alternative(11), delay(100))*, handles the exception, then state changes to *T*, execution is done. Last, rule 4.1 is applied to get conclusion in line 5 from hypothesis in line 4, since $P_{AddTwoDouble}$ =*seq(P_2, alternative(101))*, and P_2 = *repeat(P_1,2)*. This example only shows one possible execution of $P_{AddTwoDouble}$, other executions can be simulated similarly using the semantics rules.

Scientific workflows execution often involves complicated workflow structure, distributed computation resources, long time running duration. Scientific workflow design often needs to be user friendly enough to make the learning curve of using the system as low as possible. Moreover, scientific worflow management systems need to be reliable. The systems should be able to intelligent enough to deal with certain level of exceptions with the help of exception handlers in order to ensure successful execution of scientific workflows. Our proposed exception handling language bring a solution to solve exception handling issues in scientific workflow with respect to the unique natures of scientific workflow. Our language provides the ability to build complex exception handlers for different structures of workflows, both simple and nested. Traditional exception handling techniques are built in our language with a relatively simple and easy to use grammar with structural composition rules.

$$f(\vec{i}, 11, F) = T \qquad 1$$

$$(F, (\vec{i}, w, alternative(11))) \rightarrow (T, (\vec{i}, w, \phi)) \qquad 2$$

$$(F, (\vec{i}, w, seq(alternative(11), delay(100)))) \rightarrow (T, (\vec{i}, w, \phi)) \qquad 3$$

$$(F, (\vec{i}, w, repeat(P_1, 2))) \rightarrow (T, (\vec{i}, w, \phi)) \qquad 4$$

$$(F, (\vec{i}, w, P_{AddTwoDouble})) \rightarrow (T, (\vec{i}, w, \phi)) \qquad 5$$

Figure 4. Exception Handler Interpret using semantics rules of our language.

IV. Exception handler parsing and exception handling algorithms

In this section, we propose two algorithms: EHParse and HandleException. EHParse is used to parse the exception handlers specifications and HandleException is used to execute exception handlers. For each algorithm, we will first give the formal description of the algorithm and illustrate it with examples.

A. Algorithm EHParse

This algorithm takes an user designed XML format exceptoin handler specification as input, parses this exception handler specification, breaks it into symbols in our language, and outputs the expression of the exception handler. In algorithm EHParse, an exception handler specification, EHS, is modeled as an XML tree [5]. In an XML Tree T, nodes represent XML elements and edges represent parent-child relationships between XML elements. T is an ordered tree and its nodes can have attributes and values associated with them. We introduce the following notations for each element node e in T:

- e.name represents the name of e.
- e.attributes represent the set of XML attributes of e, denoted using e.a_1,,e.a_i. The names and values of these attributes by e.a_i:name and e.a_i:value, respectively (i = 1,...,n)
- e.value represents the value of e.
- e.children represents the ordered sequence of child nodes of e, denoted using e.c_1, ,e.$c_m$and e.children = NULL if e is a leaf node of T.

Figure 5 gives the detailed steps of algorithm EHParse. This algorithm reads each element in the specification, parses the specification into details with respect to our language syntax and gets the corresponding exception handler expression. The case of empty exception handlers will be checked by workflow engine before algorithm EHParse is called. $e.value means to get the value of element e.

```
00 Algorithm EHParse
01 Input: an exception handler specification, EHS
02 Output: the expression of the exception handler, exp
03 Begin
04    element e in EHS
05        If e.name == "RETRY" then
06            Return retry($e.value)
07        End If
08        If e.name == "ALTERNATIVE" then
09            Return alternative($e.value)
10        End If
11        If e.name == "DELAY" then
12            Return delay($e.value)
13        End If
14        If e.name == "SEQUENCE" then
15            exp$_1$ = EHParse(e.c$_1$) //child 1 of e
16            exp$_2$ = EHParse(e.c$_2$) //child 2 of e
17            Return seq(exp$_1$, exp$_2$)
18        End If
19        If e.name == "REPEAT" then
20            tmpexp = EHParse(e.c$_1$)
21            Return repeat(tmpexp, $e.a$_1$:value)
22        End If
23 End Algorithm
```

Figure 5. Algorithm EHParse.

Take the exception handler specification of primitive workflow *AddTwoDouble* in Figure 2 as an example. EH-Parser reads the exception handler specification and check the name of element e to get the matching case. For this example, EHParse goes to line 14 because e.name == "SE-QUENCE". Then we begin to parse the elements under SE-QUENCE using two recursive calls, exp1 = EHParse(e.c$_1$), exp2 = EHParse(e.c$_2$). Since for line 15 and 16, e.c$_1$.name == "REPEAT" and e.c$_2$.name == "ALTERNATIVE", two recursive calls will go to line 20 and line 09. For exp2, we can get exp2 = *alternative(101)*. For exp1, EHParse will parse the elements under REPEAT using another recursive call, read the elements under REPEAT element (line 20) and get the value of *n* in *repreat(P, n)*. Then we will find another SEQUENCE element and fall into the case of SEQUENCE again, because for line 21, e.c$_1$.name == "SEQUENCE". Inside this pair of SEQUENCE element, we locate two base cases: ALTERNATIVE and DELAY with their values, so tmpexp = *seq(alternative(11), delay(100))*. Then *repeat(seq(alternative(11), delay(100)), 2)* is returned as exp1. Therefore, the complete expression of the exception handler for primitive *AddTwoDouble* in Figure 2 is returned as *seq(repeat(seq(alternative(11), delay(100)), 2), alterna-tive(101))*.

B. Algorithm HandleException

This algorithm takes an exception handler expression as input, interprets this exception handler expression, and outputs the execution state of the exception handler. Figure 6 gives the detailed steps of algorithm HandleException. Function *f*: $\overrightarrow{i} \times w \times s \rightarrow s$ is used to model the execution of a particular workflow in exception handlers, value of state *s*: *T* and *F*, are used as output. We use the exception handler expression we got using the EHParse algorithm, exp = *seq(repeat(seq(alternative(11), delay(100)), 2), alter-native(101))*, as an example to illustrate this algorithm. Use HandleException algorithm, we will interpret the expression using pattern matching. The HandleException algorithm will first read seq, and falls into the pattern of *seq*(P_1, P_2) (line 25), in which P_1 = *repeat(seq(alternative(11), delay(100)), 2)*, matches pattern of *repeat(P,n)* and P_2 = *alternative(101)*, matches the pattern of *alternative(t)*. HandleException algo-rithm will execute P_1 first in a recursive call (line 26), and based on the state of P_1 to decide whether execute P_2. P_1 itself matches the case of *repeat(P,n)* (line 33), where *P* = *seq(alternative(11), delay(100))* and *n* = 2. Then another recursive call (line 36) will be made for *P*. *P* matches the *seq*(P_1, P_2) pattern, where P_1 = *alternative(11)* and P_2 = *delay(100)*. Then the pattern of *alternative(t)* and *delay(k)* are matched. Therefore, the execution of this exception handler will start from *alternative(11)*, based on its execution state, following HandleException algorithm.

```
00 Algorithm HandleException
01 Input: an exception handler expression, exp
02 Output: execution state of the exception handler, T (exception is handled)
          or F(exception is not handled)
03 Begin
04  State s := F
05   If exp == $\phi$ then Return s End If
06   Switch (exp)
07   {
08   Case retry(i):
09          Given current workflow W
10          for(int j=0; j<i;j++)
11          {
12             s = f(i, W, E)
13             If s == T then
14              Return T
15          }
16          Return s
17   Case delay(k):
18          sleep(k)
19          Return s
20   Case alternative(t):
21          s = f(i, t, E)
22          If s == T then
23             Return T
24          Else Return s
25   Case seq($P_1$,$P_2$):
26          State s1 = HandleException($P_1$)
27          If s1 == F then
28             State s2= HandleException($P_2$)
29                Return s2
30          Else
31             Return T
32          End If
33   Case repeat(P,n):
34          for(int i=0;i< n;i++)
35          {
36              s = HandleException(P)
37              If s == T then
38                Return T
39          }
40          Return s
41   }
42 End
```

Figure 6. Algorithm HandleException.

V. ARCHITECTURE DESIGN

In this section, we introduce the exception handling architecture prototype in our VIEW system. Current ver-sion of VIEW includes two major parts: the Workbench and the VIEW Kernel. The Workbench is a light weight client side where users design workflows, and the VIEW Kernel is on a server and in charge of the execution of workflows and other features. Figure 7 gives the exception handling architecture design in VIEW scientific workflow management system. The left hand side of the dash line is in the Workbench, and the right hand side is in the VIEW Kernel. Workflow design and exception handler design are done in the Workbench during workflow design time. We leave the choice to end users to decide whether an exception handler will be specified. After the design of workflows and the exception handlers is done in the Workbench, the workflow specification file is generated (exception handler specification included). The workflow execution is started

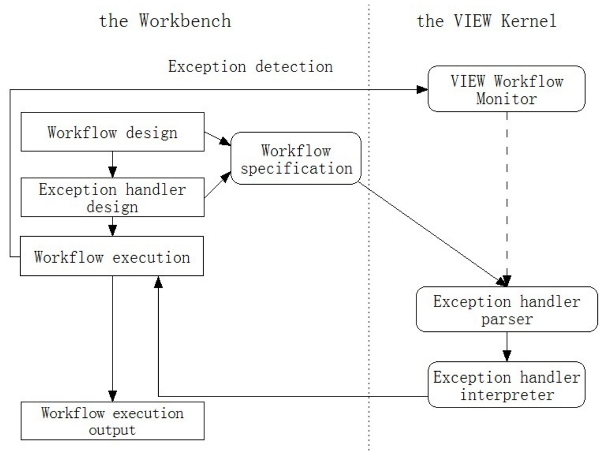


Figure 7. Exception Handling Architecture in VIEW.

in the Workbench and execution output is shown in the Workbench. Actual execution is taken care of by the VIEW Kernel and may involve distribute resources. When the end user starts the workflow execution, workflow engine in the VIEW Kernel will take action to invoke each part of the workflow and pass data from one part to another. If there is no exception during the workflow execution, the workflow execution will end with correct output data. If there are some exceptions occur, exceptions will be detected by the VIEW Workflow Monitor in the VIEW Kernel and trigger exception handling (dash line with arrow from VIEW Workflow Monitor to Exception handler parser in the VIEW Kernel). Then we start the exception handling part. Exception handler parser, which is implemented following the EHParse algorithm, will parse the specification file, and get the exception handler expression. Then this expression will be sent to the exception handling interpreter, which implements HandleException algorithm. Then the execution state for the exception handler will be updated and output of execution will be returned to the Workbench to display.

VI. Related work

Much work has been done to resolve exception handling issues in workflow management systems. A classification of exceptions in workflow management systems and a taxonomy of exception handling for workflow systems were presented in [1]. Moreover, many efforts have been made to address this problem in business workflows. In [6] [7], a framework of exception handling was introduced in forms of exception handling patterns. An implementation of these patterns was presented in [8]. These exception handling patterns were based on Petri net and used to as primitives for exception handling. Combinations of these patterns are used to form exception handling mechanism. These patterns are rigid due to underlying Petri net nature and the use of these patterns is not very efficient. Other representative work includes [9]. Recently, scientific workflows become widely used by scientists for scientific research. The exception handling issue in the context of scientific workflows has been widely recognized. In [10], two exception handling patterns were introduced to propagate and handle exceptions in hierarchical workflows. Reference net is used to represent workflows and alternative workflow is used to do exception handling. A fault-tolerant and recovery service for grid based scientific workflow was presented in [11], this approach is mainly concentrated on the exceptions from computation resources. [12] introduced a Quality of Resilience model to represent the reliability level of workflows from the resource point of view. Scientific workflow management systems are tools used to develop and execute scientific workflows. There are several representative ones, such as Taverna [13], Kepler [14], VIEW [2]. Significant work has been done in these SWFMSs for exception handling to ensure successful workflow execution. For the Kepler system, in [15], a number of typical failure patterns in scientific workflow were described, and the corresponding recovery strategies were also mentioned. In [15], a fault-tolerant architecture was introduced, as well as a framework which consists of a contingency Kepler actor, a external monitoring module. An checkpointing mechanism mentioned in [16], this check point mechanism uses provenance information. A more recent version of fault tolerant framework with implementation and performance analysis was presented in [17]. For Taverna, the exception handling features were mentioned in [18]. Some exception handling techniques, such as retry, alternative, can be configured in the processors in Taverna. In [19], syntax and semantics of Taverna workflow were presented, however, no exception handling was mentioned. For VIEW, a task level exception handling framework was presented in [20], that work focuses on the exception handling using different implementations within one primitive workflow.

None of work above tackled the exception handling in scientific workflows from the language point of view, which is the focus of this paper. Since language is system independent, our language is designed for VIEW, but it can also be extended and used by others, which is also beneficial. Our proposed language is structured and compositional. It allows end users to define exception handlers using other exception handlers. Moreover, our language provides exception handlers for both simple and nested workflows. A set of inference rules of our language semantics is introduced and can be used for reasoning execution of exception handlers. And two exception handling algorithms are also presented for implementation. Our language provides a generic solution and a formal foundation for exception handling in scientific workflow.

VII. Conclusions and future work

In this paper, we proposed a user-defined exception handling framework in the VIEW scientific workflow system. An exception handling language was presented, with both syntax and operational semantics. We presented the language syntax in two forms: abstract form using BNF, and concrete form using XML-based scientific workflow specification. Concrete examples were given to demonstrate our language syntax. We defined the operational semantics rules of our language in terms of a set of inference rules and used an example to illustrate the exception handling process. Moreover, we also introduced the exception handling architecture in VIEW as well as two algorithms to parse exception handler specification and interpret exception handler expression. Future work includes the model of outputs during exception handling to support dataflow exception handling, as well as soundness and completeness of our proposed language.

References

[1] D. Ruan and S. Lu, "Exception Handling in Workflow Systems: A Survey," TR-SWR-02-2009, Technical Report 2009.

[2] C. Lin, S. Lu, X. Fei, A. Chebotko, D. Pai, Z. Lai, F. Fotouhi, and J. Hua, "A Reference Architecture for Scientific Workflow Management Systems and the VIEW SOA Solution," *IEEE Transactions on Services Computing*, vol. 2, pp. 79–92, 2009.

[3] C. Lin, S. Lu, L. Z., A. Chebotko, X. Fei, J. Hua, and F. Fotouhi, "Service-Oriented Architecture for VIEW: A Visual Scientific Workflow Management System," in *Proceedings of IEEE International Conference on Services Computing*, vol. 1, 2008, pp. 335 –342.

[4] X. Fei and S. Lu, "A Dataflow-Based Scientific Workflow Composition Framework," *IEEE Transactions on Services Computing*, vol. 5, no. 1, pp. 45–58, 2012.

[5] M. Atay, A. Chebotko, D. Liu, S. Lu, and F. Fotouhi, "Efficient schema-based XML-to-Relational data mapping," *Information Systems*, vol. 32, no. 3, pp. 458–476, 2007.

[6] R. Nick, W. M. P. v. d. Aalst, and A. H. M. t. Hofstede, "Workflow exception patterns," in *Proceedings of 18th International Conference on Advanced Information Systems Engineering(CAiSE)*, 2006, pp. 288–302.

[7] R. Nick and A. H. M. T. Hofstede, "Exception Handling Patterns in Process-Aware Information Systems," BPM Center Report BPM-06-04 2006.

[8] M. Adams, A. H. M. Ter Hofstede, W. M. P. Van Der Aalst, and D. Edmond, "Dynamic, extensible and context-aware exception handling for workflows," in *Proceedings of the 2007 OTM Confederated international conference*, 2007, pp. 95–112.

[9] C. Hagen and G. Alonso, "Exception handling in workflow management systems," *IEEE Transactions on Software Engineering*, vol. 26, no. 10, pp. 943 –958, 2000.

[10] R. Tolosana-Calasanz, J. A. Bañares, O. F. Rana, P. Álvarez, J. Ezpeleta, and A. Hoheisel, "Adaptive exception handling for scientific workflows," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 5, pp. 617–642, 2010.

[11] G. Kandaswamy, A. Mandal, and D. Reed, "Fault Tolerance and Recovery of Scientific Workflows on Computational Grids," in *Proceedings of 8th IEEE International Symposium on Cluster Computing and the Grid*, 2008, pp. 777 –782.

[12] R. Tolosana-Calasanz, M. Lackovic, O. F. Rana, J. Bañares, and D. Talia, "Characterizing quality of resilience in scientific workflows," in *Proceedings of the 6th workshop on Workflows in support of large-scale science*, 2011, pp. 117–126.

[13] T. Oinn, M. Addis, J. Ferris, D. Marvin, T. Carver, M. R. Pocock, and A. Wipat, "Taverna: A tool for the composition and enactment of bioinformatics workflows," *Bioinformatics*, vol. 20, 2004.

[14] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao, "Scientific workflow management and the Kepler system: Research Articles," *Concurrency and Computation: Practice & Experience*, vol. 18, no. 10, pp. 1039–1065, 2006.

[15] P. Mouallem, D. Crawl, I. Altintas, M. Vouk, and U. Yildiz, "A fault-tolerance architecture for Kepler-based distributed scientific workflows," in *Proceedings of the 22nd international conference on Scientific and statistical database management*, 2010, pp. 452–460.

[16] D. Crawl and I. Altintas, "A provenance-based fault tolerance mechanism for scientific workflows," *Provenance and Annotation of Data and Processes*, pp. 152–159, 2008.

[17] M. Vouk and P. Mouallem, "On High-Assurance Scientific Workflows," in *Proceedings of IEEE 13th International Symposium on High-Assurance Systems Engineering*, 2011, pp. 73 –82.

[18] P. Missier, S. Soiland-Reyes, S. Owen, W. Tan, A. Nenadic, I. Dunlop, A. Williams, T. Oinn, and C. Goble, "Taverna, reloaded," in *Proceedings of the 22nd international conference on Scientific and statistical database management*, 2010, pp. 471–481.

[19] D. Turi, P. Missier, C. Goble, D. Roure, and T. Oinn, "Taverna workflows: Syntax and semantics," in *Proceedings of IEEE International Conference on e-Science and Grid Computing*, 2007, pp. 441–448.

[20] D. Ruan and S. Lu, "Task Exception Handling in the VIEW Scientific Workflow System," in *Proceedings of IEEE International Conference on Services Computing*, 2010, pp. 637–638.