Automatic Generation of Workflow Provenance

Roger S. Barga[1] and Luciano A. Digiampietri[2]

[1] Microsoft Research, One Microsoft Way
Redmond, WA 98052, USA
[2] Institute of Computing, University of Campinas,
Sao Paolo, Brazil
barga@microsoft.com

Abstract. While workflow is playing an increasingly important role in e-Science, current systems lack support for the collection of provenance data. We argue that workflow provenance data should be automatically generated by the enactment engine and managed over time by an underlying storage service. We briefly describe our layered model for workflow execution provenance, which allows navigation from the conceptual model of an experiment to instance data collected during a specific experiment run, and back.

1 Introduction

Many scientific disciplines today are data and information driven, and new scientific knowledge is often obtained by scientists scheduling data intensive computing tasks across an ever-changing set of computing resources. Scientific workflows represent the logical culmination of this trend. They provide the necessary abstractions that enable effective usage of computational resources, and the development of robust problem-solving environments that marshal both data and computing resources. Part of the established scientific method is to create a record of the origin of a result, how it was obtained, experimental methods used, the machines, calibrations and parameter settings, quantities, etc. It is the same with scientific workflows, except here the result provenance is a record of workflow activities invoked, services and databases used, their versions and parameter settings, data sets used or generated and so forth. Without this provenance data, the results of a scientific workflow are of limited value.

Though the value of result provenance is widely recognized, today most workflow management systems generate only limited provenance data. Consequently, this places the burden on the user to manually record provenance data in order to make an experiment or result explainable. Once the experiment is finished, the specific steps leading to the result are often stored away in a private file or notebook. However, weeks or months may pass before the scientist realizes a result was significant, by which time provenance has been forgotten or lost – it simply slips through the cracks.

We believe the collection of provenance data should be automatic, and the resulting provenance data should be managed by the underlying system. Moreover, a robust provenance trace should support multiple levels of representations. In some cases, it is suitable to provide an abstract description of the scientific process that took place, without describing specific codes executed, the data sets or remote services

L. Moreau and I. Foster (Eds.): IPAW 2006, LNCS 4145, pp. 1 – 9, 2006.

invoked. In other cases, a precise description of the execution of the workflow, with all operational details such as parameters and machine configurations provided, is exactly what is required to explain a result. Given no single representation can satisfy all provenance queries, the system should generate multiple related representations simultaneously. The ideal model will allow users to navigate from the abstract levels into lower levels and map the latter backwards to higher level abstractions. This also allows scientists to specify exactly what they wish to share from their experiment.

In this paper, we suggest the workflow enactment engine should be responsible for generating workflow provenance *automatically* during runtime, and an underlying storage service should be responsible for managing workflow provenance data. Provenance collection and management should happen transparently. That is, users should not have to take any special actions or execute special commands to have provenance of their workflow execution collected and maintained. Indeed, unless users take extraordinary action, such as purging an experiment result, an accurate provenance record will be maintained for results residing on their system. By making the enactment system responsible for the creation and collection of execution provenance, we are able to generate provenance automatically, freeing users from having to manually track provenance during execution. Equally important, when properly collected and maintained, workflow provenance data can be indexed and queried as a *first class data product* using conventional data management tools and techniques. In this paper, we also outline a *layered* or *factored* model for representing workflow execution provenance, from an abstract description of the workflow (scientific process) to a precise description of execution level details for a single experiment, which enables provenance data to be linked and defined in a way for more effective discovery, integration and cooperation.

2 Workflow Execution Provenance Model

In this section we outline the machine-readable model we propose to represent provenance data captured during workflow execution. The attractions of creating a machine-readable model of an experiment, that is, a workflow, are not difficult to envisage. Three key benefits that a *multilayered model* can bring are explanation, validation, and insight. Having an abstract model of the science being undertaken, and being able to use the model to interpret behavior (services and data sets used), together make it possible to explain or reason about the science. For validation, knowing which individual activities were executed, identifying branches taken during execution, steps skipped or inserted, and specific parameters supplied at runtime is essential when trying to establish trust in a result. The ability to compare a set of related experiment executions against an abstract workflow model, to identify common patterns or options not yet explored, can be used to gain insight in authoring new experiment designs. A related benefit, not discussed in this short paper is optimization – a layered or *factored* model can expose opportunities to efficiently store provenances traces in the underlying storage manager.

The first level in our model, illustrated in Figure 1 below, represents an abstract description of the scientific process. This layer captures *abstract activities* in the workflow and *links/relationships* among the activities or ports of activities, where an abstract activity is a *class* of activities. These classes can be described by the function of the activity (e.g., InvokeWebService, or Sequential, Parallel Branch and While activities, etc) or through semantic description – for example, a class of *alignment* activities, where Blastx and FASTA are instances that belong to this class.

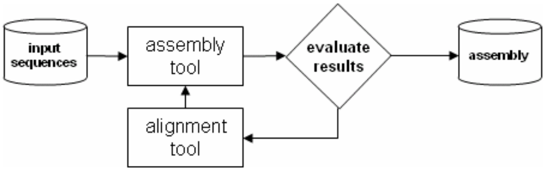


Fig. 1. Level L0, represents abstract service descriptions

The second level in our model, illustrated in Figure 2 below, represents an instance of the abstract model. This layer captures bindings or *instances of activities* and additional *relationships*, as classes of activities are instantiated. This level refines abstract activities specified in the previous level with a specific instance (for example, *InvokeWebService* is bound to a specific activity instance *InvokeBlastWebService* and *Alignment tool* is bound to *Blastx tool*. The additional information captured at this level is required *to* assign *fields of the classes* of the previous level. For example, the *InvokeWebService* has fields *ProxyClass* and *MethodName* that should be filled, with a proxy where the web service can be found and name of the method to be invoked.

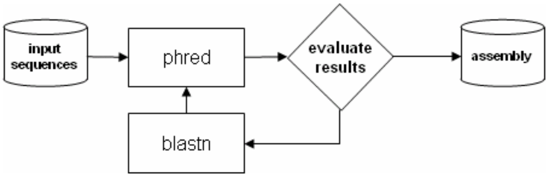


Fig. 2. Level L1, represents service instantiation descriptions

In the next level of the model L2, illustrated in Figure 3, represents the execution of the workflow instance specified in level L1. From level L1 to level L2 the model captures information provided at runtime, such as *parameters* and properties that complete the specification of activities, including input data, runtime parameter supplied, activities inserted or skipped during execution, etc. Information captured at

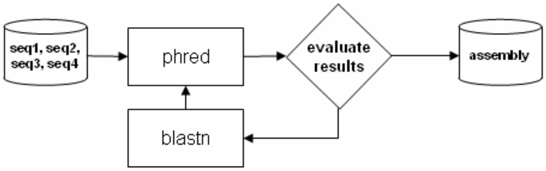


Fig. 3. Level L2, represents data instantiation of the executable workflow

this level is sufficient to trace the execution of the workflow, from input parameters to individual activities executed and runtime parameters supplied to each activity.

The final level of our model L3, illustrated in Figure 4 below, represents runtime specific information. Here we capture operational specific details, such as the start and end time of the workflow execution, the start and end time of the individual activity execution, status codes and intermediary results, information about the internal state of each activity, along with information about the machines to which each activity was assigned for execution.

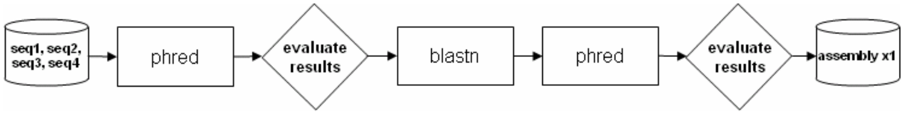
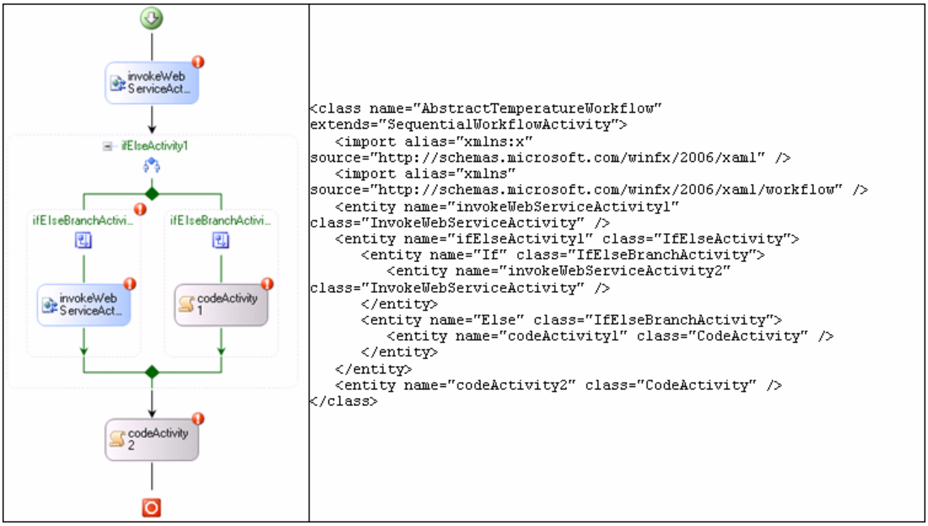


Fig. 4. Level L3, run-time provenance for a workflow execution

At each level of our model, we extend the amount of detail and amount of information of the previous level. The linkage between levels is facilitated by three types of operations: *extend* (class extension); *instance* (class instantiation) and *import* (reference to a specific model). Resuming, in level 0 the user designs a class from a model that is extended in level 1. In the level 2, the class of level 1 is instantiated. Level 3 imports the model of level 2 to make references to its entities.

3 Basic Workflow Execution Provenance Sample

In this section we illustrate the information that can be captured in our model using a simple example. Our implementation is based on the Windows Workflow Foundation



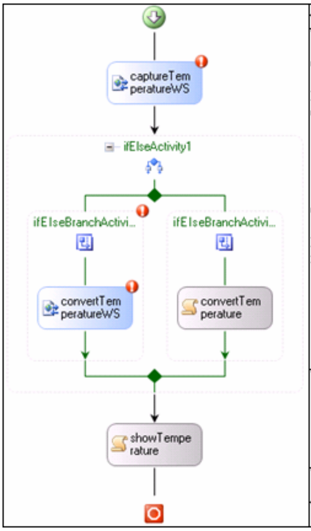
```
<class name="AbstractTemperatureWorkflow"
extends="SequentialWorkflowActivity">
   <import alias="xmlns:x"
source="http://schemas.microsoft.com/winfx/2006/xaml" />
   <import alias="xmlns"
source="http://schemas.microsoft.com/winfx/2006/xaml/workflow" />
   <entity name="invokeWebServiceActivity1"
class="InvokeWebServiceActivity" />
   <entity name="ifElseActivity1" class="IfElseActivity">
      <entity name="If" class="IfElseBranchActivity">
         <entity name="invokeWebServiceActivity2"
class="InvokeWebServiceActivity" />
      </entity>
      <entity name="Else" class="IfElseBranchActivity">
         <entity name="codeActivity1" class="CodeActivity" />
      </entity>
   </entity>
   <entity name="codeActivity2" class="CodeActivity" />
</class>
```

Fig. 5. Workflow level L0

(WinFX) [1], an extensible framework for developing workflow solutions. WinFX has a predefined set of activities (If-Else, While, Parallel, InvokeWebService, etc) and allows for user-defined custom activities by object inheritance from base classes.

In level L0 the user defines type of workflow (*Sequential*, *StateMachine* or *Rule Driven*) she wants to construct and identifies classes of activities available for use. Fig. 5 shows the graphical representation of the workflow and corresponding code. The data stored in the provenance for this level supports general queries, such as: what experiments in my collection use web services? (*invokeWebServiceActivities*) or what experiments utilize alignment tools? This level, in general, describes the design space for all workflows that are an instance of this abstract model.

In level L1 the user replaces abstract activities with concrete activities, and enters fields pre-defined for the individual abstract activities. For example, the activity *invokeWebServiceActivity1* of Figure 5 was specialized to *captureTemperatureWS* in Fig. 6 and the property *ProxyClass* was filled with the desired proxy. Another property that was filled in was *MethodName*, with the name of the method from the Web Service the user wants to invoke. The provenance model of L2 models a specific instance of the abstract workflow model that is being prepared for execution.

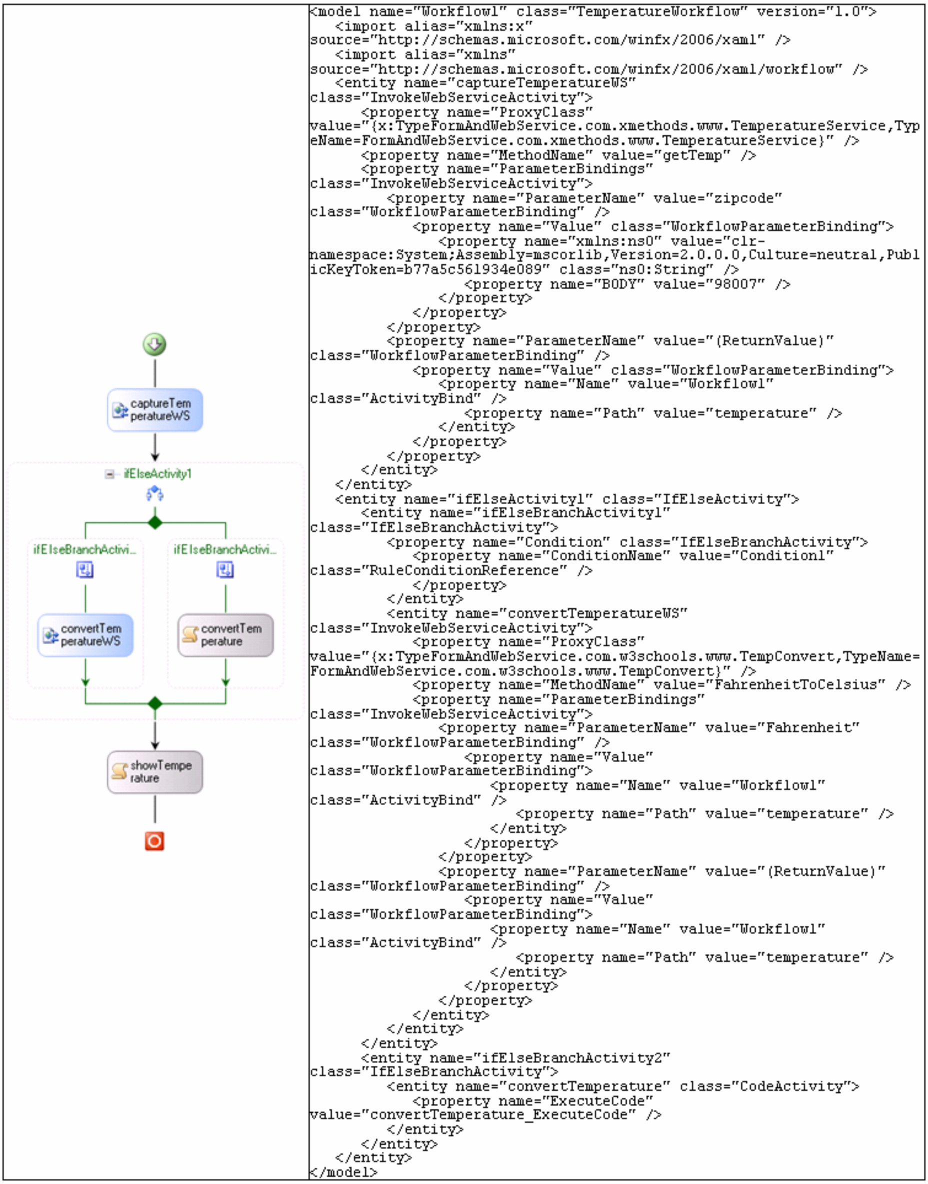


```
<class name="TemperatureWorkflow" extends="AbstractTemperatureWorkflow"
version="1">
   <entity name="captureTemperatureWS"
class="InvokeWebServiceActivity">
      <property name="ProxyClass" value="{x:Type
FormAndWebService.com.xmethods.www.TemperatureService,TypeName=FormAndW
ebService.com.xmethods.www.TemperatureService}" />
      <property name="MethodName" value="getTemp" />
   </entity>
   <entity name="ifElseActivity1" class="IfElseActivity">
      <entity name="If">
         <entity name="convertTemperatureWS"
class="InvokeWebServiceActivity">
            <property name="ProxyClass" value="{x:Type
FormAndWebService.com.w3schools.www.TempConvert,TypeName=FormAndWebServ
ice.com.w3schools.www.TempConvert}" />
            <property name="MethodName" value="FahrenheitToCelsius" />
         </entity>
      </entity>
      <entity name="Else">
         <entity name="convertTemperature" class="CodeActivity">
            <property name="ExecuteCode"
value="convertTemperature_ExecuteCode"/>
         </entity>
      </entity>
   </entity>
   <entity name="showTemperature" class="CodeActivity">
      <property name="ExecuteCode"
value="showTemperature_ExecuteCode"/>
   </entity>
</class>
```

Fig. 6. Workflow level L1

From the provenance in Level L1 the user can pose queries about specific activities: What experiments (workflows) used the web service *TemperatureService*? Or, what experiments invoke the following method of a service, or what experiments use the same alignment tool?

In the level L2, which is the executable workflow, all data sets are identified and data parameters are supplied so that the workflow is ready to be executed. One example of parameter that was filled is *zipcode* from *captureTemperatureWS* that has the value “98007” (see more details in Fig. 7). In this level of the model, we can



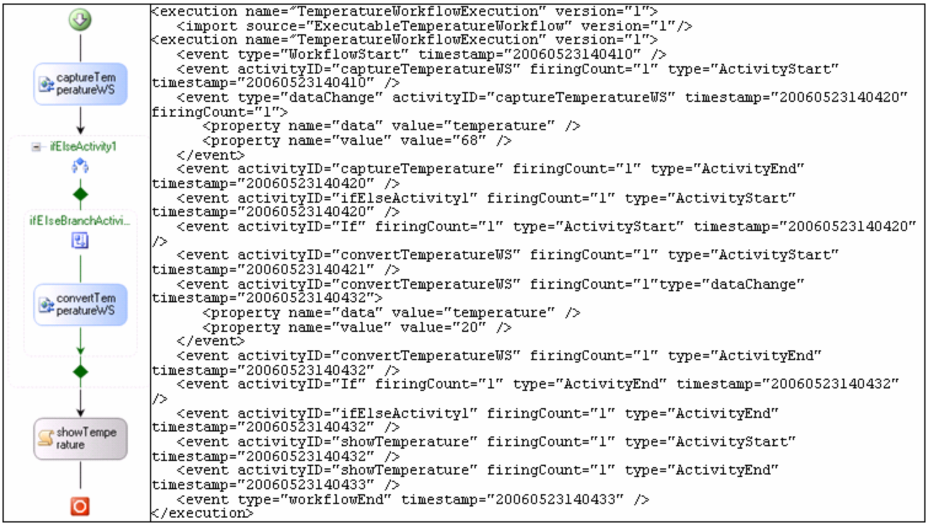
```
<model name="Workflow1" class="TemperatureWorkflow" version="1.0">
   <import alias="xmlns:x"
source="http://schemas.microsoft.com/winfx/2006/xaml" />
   <import alias="xmlns"
source="http://schemas.microsoft.com/winfx/2006/xaml/workflow" />
   <entity name="captureTemperatureWS"
class="InvokeWebServiceActivity">
      <property name="ProxyClass"
value="{x:TypeFormAndWebService.com.xmethods.www.TemperatureService,Typ
eName=FormAndWebService.com.xmethods.www.TemperatureService}" />
      <property name="MethodName" value="getTemp" />
      <property name="ParameterBindings"
class="InvokeWebServiceActivity">
         <property name="ParameterName" value="zipcode"
class="WorkflowParameterBinding" />
            <property name="Value" class="WorkflowParameterBinding">
               <property name="xmlns:ns0" value="clr-
namespace:System;Assembly=mscorlib,Version=2.0.0.0,Culture=neutral,Publ
icKeyToken=b77a5c561934e089" class="ns0:String" />
                  <property name="BODY" value="98007" />
               </property>
            </property>
         </property>
         <property name="ParameterName" value="(ReturnValue)"
class="WorkflowParameterBinding" />
            <property name="Value" class="WorkflowParameterBinding">
               <property name="Name" value="Workflow1"
class="ActivityBind" />
                  <property name="Path" value="temperature" />
               </entity>
            </property>
         </property>
      </entity>
   </entity>
   <entity name="ifElseActivity1" class="IfElseActivity">
      <entity name="ifElseBranchActivity1"
class="IfElseBranchActivity">
         <property name="Condition" class="IfElseBranchActivity">
            <property name="ConditionName" value="Condition1"
class="RuleConditionReference" />
            </property>
         </entity>
         <entity name="convertTemperatureWS"
class="InvokeWebServiceActivity">
            <property name="ProxyClass"
value="{x:TypeFormAndWebService.com.w3schools.www.TempConvert,TypeName=
FormAndWebService.com.w3schools.www.TempConvert}" />
            <property name="MethodName" value="FahrenheitToCelsius" />
            <property name="ParameterBindings"
class="InvokeWebServiceActivity">
               <property name="ParameterName" value="Fahrenheit"
class="WorkflowParameterBinding" />
                  <property name="Value"
class="WorkflowParameterBinding">
                     <property name="Name" value="Workflow1"
class="ActivityBind" />
                        <property name="Path" value="temperature" />
                     </entity>
                  </property>
               </property>
               <property name="ParameterName" value="(ReturnValue)"
class="WorkflowParameterBinding" />
                  <property name="Value"
class="WorkflowParameterBinding">
                     <property name="Name" value="Workflow1"
class="ActivityBind" />
                        <property name="Path" value="temperature" />
                     </entity>
                  </property>
               </property>
            </entity>
         </entity>
      </entity>
      <entity name="ifElseBranchActivity2"
class="IfElseBranchActivity">
         <entity name="convertTemperature" class="CodeActivity">
            <property name="ExecuteCode"
value="convertTemperature_ExecuteCode" />
         </entity>
      </entity>
   </entity>
</model>
```

Fig. 7. Workflow level L2

pose a number of queries which do not involve runtime information. These queries can involve parameters, activities and abstract activities/structures. What workflows used the value “98007” in *zipcode* parameters? What was the precondition for the workflow follow the path *If* in the *IfThenElseActivity1*?

The last level (runtime level) can contain a great diversity of information. The kind of information to be stored will depend of the goals of the user. For example, in a system that utilizes a high performance Grid, the user may wish to know what activity (job) was executed in each processor of the Grid and when this execution occurred. In other systems, the user may wish to store information about the internal status of each activity. These two examples are allowed by our provenance model, but require changes in the applications to work (in the Grid example, the scheduler must send information about the process to the provenance system, and in the second example each activity must send information to the provenance system).

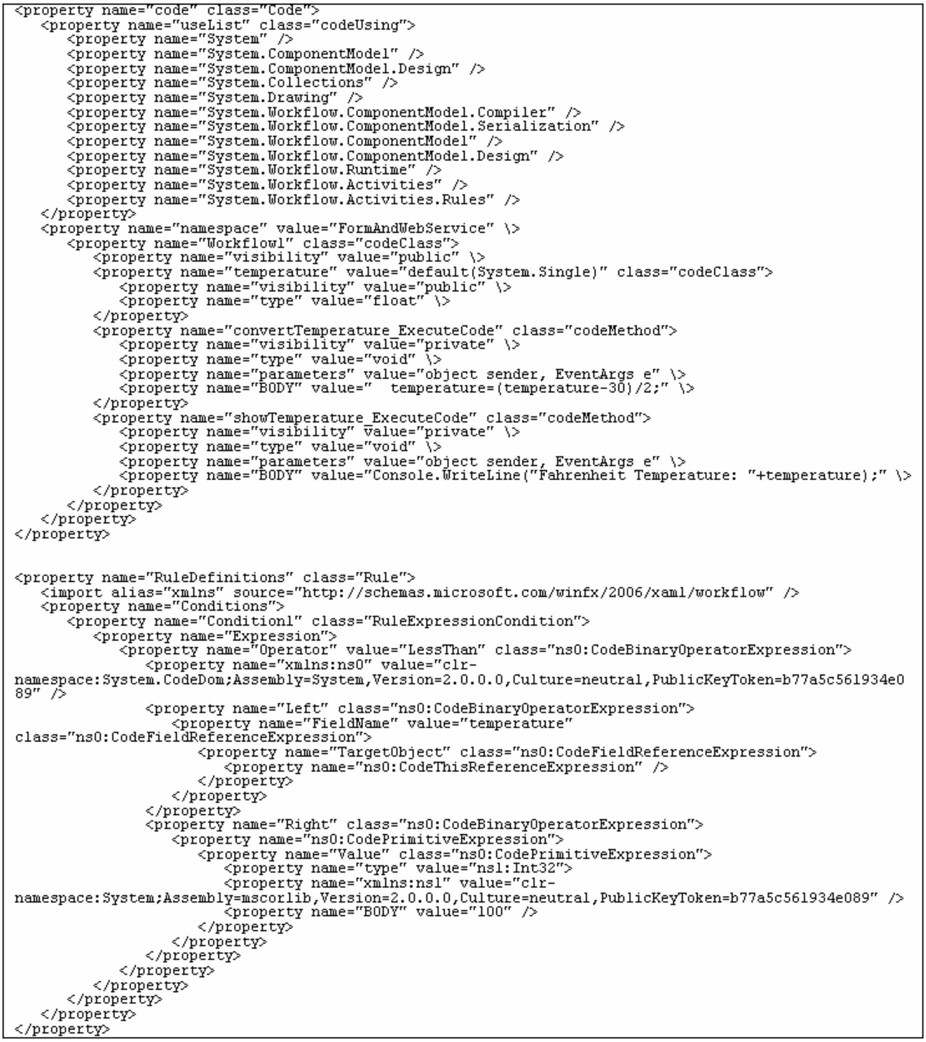
Fig. 8 shows a simple runtime record from the Executable Workflow of Fig. 7. In this example, only information about start and end time of the workflow and of the activities were stored and information about produced data.



```
<execution name="TemperatureWorkflowExecution" version="1">
   <import source="ExecutableTemperatureWorkflow" version="1"/>
<execution name="TemperatureWorkflowExecution" version="1">
   <event type="WorkflowStart" timestamp="20060523140410" />
   <event activityID="captureTemperatureWS" firingCount="1" type="ActivityStart"
timestamp="20060523140410" />
   <event type="dataChange" activityID="captureTemperatureWS" timestamp="20060523140420"
firingCount="1">
      <property name="data" value="temperature" />
      <property name="value" value="68" />
   </event>
   <event activityID="captureTemperature" firingCount="1" type="ActivityEnd"
timestamp="20060523140420" />
   <event activityID="ifElseActivity1" firingCount="1" type="ActivityStart"
timestamp="20060523140420" />
   <event activityID="If" firingCount="1" type="ActivityStart" timestamp="20060523140420"
/>
   <event activityID="convertTemperatureWS" firingCount="1" type="ActivityStart"
timestamp="20060523140421" />
   <event activityID="convertTemperatureWS" firingCount="1"type="dataChange"
timestamp="20060523140432">
      <property name="data" value="temperature" />
      <property name="value" value="20" />
   </event>
   <event activityID="convertTemperatureWS" firingCount="1" type="ActivityEnd"
timestamp="20060523140432" />
   <event activityID="If" firingCount="1" type="ActivityEnd" timestamp="20060523140432"
/>
   <event activityID="ifElseActivity1" firingCount="1" type="ActivityEnd"
timestamp="20060523140432" />
   <event activityID="showTemperature" firingCount="1" type="ActivityStart"
timestamp="20060523140432" />
   <event activityID="showTemperature" firingCount="1" type="ActivityEnd"
timestamp="20060523140433" />
   <event type="workflowEnd" timestamp="20060523140433" />
</execution>
```

Fig. 8. Workflow level 3

Note that in workflows designed using the Windows Workflow Foundation system there are no ports/channels to data exchange. This exchange is made via workflow global variables. Our provenance model allows for this information (and information about the code of each *CodeActivity*) be stored inside the workflow model (in level L2). The rules of the workflow also can be stored in the *ExecutableWorkflow* level. Fig. 9 shows the Code and Rules for the workflow *ExecutableWorkflow* in Fig. 7. Together, these are sufficient to recreate the original workflow specification.



```
<property name="code" class="Code">
   <property name="useList" class="codeUsing">
      <property name="System" />
      <property name="System.ComponentModel" />
      <property name="System.ComponentModel.Design" />
      <property name="System.Collections" />
      <property name="System.Drawing" />
      <property name="System.Workflow.ComponentModel.Compiler" />
      <property name="System.Workflow.ComponentModel.Serialization" />
      <property name="System.Workflow.ComponentModel" />
      <property name="System.Workflow.ComponentModel.Design" />
      <property name="System.Workflow.Runtime" />
      <property name="System.Workflow.Activities" />
      <property name="System.Workflow.Activities.Rules" />
   </property>
   <property name="namespace" value="FormAndWebService" \>
      <property name="Workflow1" class="codeClass">
         <property name="visibility" value="public" \>
         <property name="temperature" value="default(System.Single)" class="codeClass">
            <property name="visibility" value="public" \>
            <property name="type" value="float" \>
         </property>
         <property name="convertTemperature_ExecuteCode" class="codeMethod">
            <property name="visibility" value="private" \>
            <property name="type" value="void" \>
            <property name="parameters" value="object sender, EventArgs e" \>
            <property name="BODY" value="  temperature=(temperature-30)/2;" \>
         </property>
         <property name="showTemperature_ExecuteCode" class="codeMethod">
            <property name="visibility" value="private" \>
            <property name="type" value="void" \>
            <property name="parameters" value="object sender, EventArgs e" \>
            <property name="BODY" value="Console.WriteLine("Fahrenheit Temperature: "+temperature);" \>
         </property>
      </property>
   </property>
</property>


<property name="RuleDefinitions" class="Rule">
   <import alias="xmlns" source="http://schemas.microsoft.com/winfx/2006/xaml/workflow" />
   <property name="Conditions">
      <property name="Condition1" class="RuleExpressionCondition">
         <property name="Expression">
            <property name="Operator" value="LessThan" class="ns0:CodeBinaryOperatorExpression">
               <property name="xmlns:ns0" value="clr-
namespace:System.CodeDom;Assembly=System,Version=2.0.0.0,Culture=neutral,PublicKeyToken=b77a5c561934e0
89" />
               <property name="Left" class="ns0:CodeBinaryOperatorExpression">
                  <property name="FieldName" value="temperature"
class="ns0:CodeFieldReferenceExpression">
                     <property name="TargetObject" class="ns0:CodeFieldReferenceExpression">
                        <property name="ns0:CodeThisReferenceExpression" />
                     </property>
                  </property>
               </property>
               <property name="Right" class="ns0:CodeBinaryOperatorExpression">
                  <property name="ns0:CodePrimitiveExpression">
                     <property name="Value" class="ns0:CodePrimitiveExpression">
                        <property name="type" value="ns1:Int32">
                        <property name="xmlns:ns1" value="clr-
namespace:System;Assembly=mscorlib,Version=2.0.0.0,Culture=neutral,PublicKeyToken=b77a5c561934e089" />
                        <property name="BODY" value="100" />
                     </property>
                  </property>
               </property>
            </property>
         </property>
      </property>
   </property>
</property>
```

Fig. 9. Rules and Code of the workflow from Fig. 7

4 Discussion

All experimental scientists know that keeping a good lab notebook is vital. This is also true for the computational scientist conducting experiments using scientific workflow systems. We believe the collection of provenance data should be automatic, and the resulting provenance data should be managed by the underlying system. In addition, we argue that a robust provenance trace should support multiple levels of representations. This model will allow users to navigate from the abstract levels into lower detailed provenance levels, depending on the amount of detail required to validate a result. A multilevel representation would permit scientists to specify exactly what they wish to share, or retain, from their experiment record.

The work presented here is very much an initial investigation in an area of growing important for e-Science. Our layered provenance model for workflow execution is a prototype to be used as a tool to explore user requirements, and to consider system requirements for managing provenance data. We envisage this provenance data to be a *first class* data resource in its own right, allowing users to both query experiment holdings to establish trust in a result and to drive the creation of future experiments.

5 Related Work

The challenge of workflow execution provenance is growing in both importance and as a research topic. Most existing provenance systems available today can be classified into one of two broad categories: i) Flow based provenance systems and ii) Annotation based provenance systems.

Flow Based Provenance Systems: In many existing Grid/Service workflow based systems, the provenance recorded is generally a variant of the following: when a service is invoked, the workflow manager stores the input, output and the service name in a provenance repository. Also, logically related service invocations are grouped by using a common ID. One disadvantage with this schema is that it can sub-optimal for running provenance based queries, and precludes the ability to compress or reuse provenance data records. And, if not properly designed it can double the depth of the provenance trace, as traversals first have to lookup the process, then lookup inputs to the process to get to the parent file. Systems that have a workflow based provenance include: PASOA [2], Chimera [3], and myGrid [4].

Annotation Based Provenance Systems: Systems such as the storage resource broker (SRB) [5] and GeoDise [6] store provenance store provenance in name-value attribute pairs. While this schema can be used to build provenance trace, the construction again can be very inefficient for the same reasons as in workflow based provenance systems, and it does not lend itself to a layered model.

References

[1] Windows Workflow Foundation (WinFX), http://msdn.microsoft.com/**workflow**/.

[2] PASOA, http://twiki.pasoa.ecs.soton.ac.uk/bin/view/PASOA/WebHome.

[3] I. Foster, J. Voeckler, M. Wilde, and Y. Zhao. The Virtual Data Grid: A New Model and Architecture for Data-Intensive Collaboration. In CIDR, January. 2003.

[4] J. Zhao, M. Goble, C.and Greenwood, C. Wroe, and R. Stevens. Annotating, linking and browsing provenance logs for e-science.

[5] M. Wan, A. Rajasekar, and W. Schroeder. Overview of the SRB 3.0: the Federated MCAT. http://www.npaci.edu/ DICE/SRB/FedMcat.html, September 2003.

[6] S. Cox, Z. Jiao, and J. Wason. Data management services for engineering. 2002.