ISSN 2278-5612

A Scheme for Generating Provenance-aware Applications through UML

Badharudheen P[1], Anu Mary Chacko[2], Dr. Madhukumar S D[3]
Department of Computer Science and Engineering, NIT Calicut, Kerala, India
[1]badharu@gmail.com
[2]anu.chacko@nitc.ac.in
[3]madhu@nitc.ac.in

ABSTRACT

The metadata that captures information about the origin of data is referred to as data provenance or data lineage. The provenance of a data item captures information about the processes and source data items that lead to its creation and its current representation. An application is said to be provenance-aware if while its execution, it captures and stores sufficient provenance information to answer provenance related queries later. Provenance information is very useful when we need to know the inter dependency of data to find error propagation or information flow. Provenance also helps us to understand the difference in two identical workflows with same inputs but different outputs. Currently, most of the provenance systems are domain specific. Through this paper, we propose a general methodology for making an application provenance-aware using basic UML design diagrams. As a starting point we have analyzed UML Class diagrams to generate information to make the application provenance-aware

Indexing terms/Keywords

UML Class Diagrams, UML Object Diagrams, Provenance.

Academic Discipline And Sub-Disciplines

Computer Science and Engineering :Software Engineering

SUBJECT CLASSIFICATION

Software Analysis and Development

TYPE (METHOD/APPROACH)

Prototype Evaluation

Council for Innovative Research

Peer Review Research Publishing System

Journal: International Journal of Management & Information Technology

Vol. 6, No. 3
editor@cirworld.com
www.cirworld.com, member.cirworld.com

ISSN 2278-5612

1. INTRODUCTION

The word ‘provenance’ very much used by art historians, was derived from the French word ‘provenir’ which means the origin, the source, or the history of the ownership or location of an object. Importance of provenance in art is that a particular piece of art will become expensive only if it is proved that it is original and genuine. Similarly in information technology, with dynamic growth of internet, it is very important to have proof to qualify the data. In the context of information technology we define provenance of an entity as the process that led to that entity being in that state.[1]

Provenance captures information about an object and the process of creation or modification of the same. The concept of provenance and its utility has recently become increasingly prominent within the domain of information processing systems.

Researchers that use data from public databases or from any other sources are very much interested in knowing about the how the data was derived. In areas like eScience and grid computing provenance information can help in reproducing results obtained or in throwing light in what went different when two identical runs of experiment with same data gave different results. Cloud computing is gaining much importance as a data processing platform due to the business advantage it proposes in the form of ‘pay as you go’ and ‘provision as you need’ models. The major concern which will hinder cloud adoption is the users’ apprehension about what happens to data while it resides in a third party location. Efficient provenance capture and management mechanism can be a good solution to this problem.

As provenance is a metadata, it is very important to have efficient means for storing, capturing and managing the same. Currently most of the work that has been done is in creating domain specific provenance solutions. In this paper we propose a generalized approach to design provenance-aware applications by adapting design in UML to provenance aware UML designs.

To determine the provenance of an item in a system, adequate documentation must be recorded while data item is being modified. The UML design diagrams of an application provide information about the structure and behaviour of the application. Careful analysis of the diagrams can give us valuable insights on the optimal locations for capturing process documentation while the application is being executed. We propose presenting this information to application developers and work with them to identify the data items for which provenance is important and discover adaptation points in the application to capture provenance.

2. Related work

There are two basic views of provenance. The first one describes the provenance of a data item as the processes that lead to its creation and the other one focuses on the source data from which the data item was derived.

In this section we provide some of the related work on provenance in the scientific and business domains.

The literatures on provenance can be broadly classified into four categories: fine granularity provenance systems, domain specific provenance systems, middleware provenance systems, and provenance in database systems.

2.1 Fine Granularity Provenance Systems

The granularity of documentation means how detailed the documentation of a process is. If a system records all the instructions in a program, whereas another system records the name of the program being run, the first system will record documentation at a finer granularity. With finer granularity documentation, the corresponding representation of provenance for a result can be more detailed. One example is the Transparent Result Caching (TREC) prototype as explained in paper [1]. TREC uses the Solaris UNIX proc system to intercept various UNIX system calls in order to build a dependency map.

A more comprehensive system which captures provenance is *audit facilities* designed for the S language. S is an interactive system for statistical analysis. The result of users commands are automatically recorded in an audit file. These results include the modification or creation of data objects as well as the commands themselves.

2.2 Domain Specific Provenance Systems

Much of the research into provenance has come in the context of domain specific applications. One major domain where provenance tools are being developed is bioinformatics. The myGrid project as explained in the paper [2] has implemented a system for recording the documentation of process in the context of in-silico experiments.

GIS is the another domain where provenance information can be recorded. Knowing the provenance of map products is critical in GIS applications because it allows one to determine their quality. Through the paper [3], Lanter developed a system for recording process documentation and retrieving the provenance of map products in a GIS.

2.3 Middleware Provenance Systems

One of the major middleware provenance systems is Chimera Virtual Data System [4], which combines a virtual data catalogue, for representing data derivation procedures and derived data, with a virtual data language interpreter that

ISSN 2278-5612

translates user requests into data definition and query operations on the database. Using the virtual data language, a user can query the catalogue to retrieve the Directed Acyclic Graph (DAG) of transformations which led to the result.

2.4 Provenance in Database Systems

In paper [5], Peter Buneman et al. define data provenance in the context of database systems as the description of the origins of data and the process by which it arrived at the database. "Why-provenance" refers to why a piece of data is in the database, i.e. what data sets (tuples) contributed to a data item, whereas, "where-provenance" addresses the location of a data element in the source data. Based on this terminology, a formal model of provenance is developed applying to both relational and XML databases.

Laura Chiticariu et al. define another model named DBNotes [6], a Post-It note system for relational databases where every piece of data may be associated with notes (or annotations). These annotations are transparently propagated along as data is being transformed. The method by which annotations are propagated is based on provenance: the annotations associated with a piece of data 'd' in the result of a transformation consist of the annotations associated with each piece of data in the source where 'd' is copied from.

In order to bring standardization in the area of provenance four provenance challenges was conducted from May 2006. As an outcome of challenges, Open Provenance Model was proposed. The goal of OPM was to define and represent provenance in a technology agnostic manner such that it can be exchanged between systems so that developers can build tools on top of it.

Related work for provenance in software engineering was done by Groth et al, in proposing a software engineering methodology, PrIMe, to make an application provenance aware. Provenance Integrating Methodology consists of three phases . First phase is to identify the provenance usecases applicable to the application. The aim of this phase is to identify the data items for which provenance information is needed. The second phase is to decompose applications into actors and map the dataflow within application. The actors are decomposed till they become “knowledgeable actors” i.e they are able to provide information about the data items identified in phase 1. Some actors may not be decomposable, this is noted separately. Once the knowledgeable actors are identified the method proposes to add wrappers to capture provenance information at that point. This scheme is giving step by step guidance to making applications which have a SOA architecture provenance aware.

3. OUR APPROACH OF MAKING AN APPLICATION PROVENANCE-AWARE USING XML

UML is a general purpose modelling language to visualise, specify, construct and document software systems. UML provides us with modelling options for all the different phases of software development. So by critically evaluating the design diagrams of the applications, we will be able to identify the modifications needed for application to make it provenance-aware. UML 2.2 provides us with two sets of diagram – one capturing the details about the structure of the application and other capturing details about the behaviour of application under different scenarios.

In this work the basic assumption the proposed method makes is that scope of the method is limited to object oriented applications being developed using UML as modelling language.

The proposed method for making an application provenance-aware uses the following additional assumptions.

- Applications are composed of objects and their interactions.
- Objects have attributes and their own operations or methods.
- Each object communicates with other objects by exchanging messages or through interactions.
- The design diagrams of the application should be developed by using any standard tools like ArgoUML, Umbrello, etc.
- Also the scope of provenance collection is limited to details captured in UML.

As part of our work we identified the basic provenance requirements to be captured by application. They are listed below.

1. How the data was created?
2. Where the data has come from?
3. Time and causal dependency
4. Dependency on external environment beyond application
5. Why the data is at this current state

We can modify the application during design to capture all these information details except requirement 4. The external environment to a great extend is determined only at the deployment of the application. We follow the three phases listed below to identify the provenance adaptation points from UML design diagram.

3.1 Design Analysis

In the design analysis phase, the UML design diagrams are analysed to identify the information needed for making application provenance-aware. By analysing the UML class diagram of an application, we can identify the different

attributes and operations or method calls of each class in the application. These method calls or interactions are the stimuli for any operation in the application.

By analysing the initial object diagram, identify the different objects and the initial states of each object. These object states will be changed later when actual interaction happens among the objects. If we are making the design diagrams of the application by using any standard design tools, we can easily parse the output design diagram with the help of an *XMI-Parser*. This *XMI-Parser* is responsible for recording the initial process documentation details from the design diagrams.

3.2 System Initialisation

In the system initialisation phase, classification of the information obtained from the diagram takes place. Using the information in data store we can work with end users of the system to model provenance questions and decide on data item for which provenance is important. In this work we have used class diagrams and object diagrams. Class diagrams provide us with details of the attributes and different operations that happen in the application. From the class diagram we populate three stores named *Opstore*, *Objstore* and *Provenance Store*. *Opstore* captures details of all the operations and *Objstore* captures the details of all the objects. The Provenance Store captures the attribute values or the states of each objects identified during runtime. The *XMI-Parser* is responsible for populating the first two stores and provenance store is populated by the application tracer.

3.3 Provenance Collection

During run time the actual provenance information is collected and recorded in the Provenance Store. For this purpose the software developer need to create an Application Tracer which tracks the interactions among the objects in the application. This *Application Tracer* is to be developed by the application developer as part of the application development. The inputs for the *Application Tracer* are taken from the *Opstore*. From the system initialization phase, we have identified the attributes for which provenance has to be collected. For each of such entries in the *Opstore*, application tracer should be designed to record the modification details in the Provenance Store. If a new object is created, it should update the *Objstore* and the Provenance Store with the new object details. Figure 1 below shows the block diagram of the proposed method.

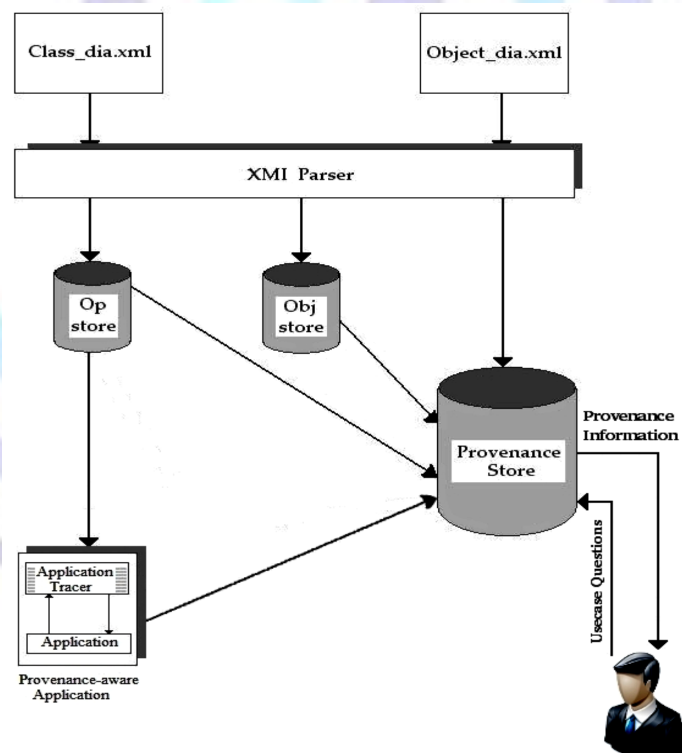

Figure 1 : Block diagram for the proposed method

3.4 THE XMI-PARSER

The XML Metadata Interchange (XMI) [8] is an Object Management Group (OMG) standard for exchanging metadata information via Extensible Markup Language (XML). The most common use of XMI is as an interchange format for UML models. Several UML tools like ArgoUML, Umbrello, etc. provide support for XMI files by importing or exporting their UML models in the XMI format.

Our *XMI-Parser* has two main functions. The first function is to define the internal structure of the three permanent stores used in the design. The second function is to parse the class diagram and the object diagram which we have given as

inputs. The *XMI-Parser* then collects the initial process documentation details such as class names, initial objects with attribute states, different operations in the application, etc. by parsing the two design diagrams. It then records this process documentation details into the three permanent stores just defined before.

The *XMI-Parser* checks for certain predefined tags that are used to define the structure of the design diagram of an application. Here each component in the design diagram has a tag with some id which can be used to identify uniquely from other components in the diagram. These tags also have some attributes with values that are used to define the property of the components. Figure 2 below shows a sample code snippet from an xmi file of a class diagram.

```
68 <UML:Class xmi.id = '127-0-1-1--172131be:13ab06c7851:87B'
69   name = 'Order' visibility = 'public' isSpecification = 'false'
70   isLeaf = 'false' isAbstract = 'false' isActive = 'false'>
71   <UML:Classifier.feature>
72    <UML:Attribute xmi.id = '127-0-1-1--172131be:13ab06c7851:87C'
73      name = 'date' visibility = 'public' isSpecification = 'false'
74      changeability = 'changeable' targetScope = 'instance'>
75      <UML:StructuralFeature.multiplicity>
76       <UML:Multiplicity xmi.id = '127-0-1-1--172131be:13ab06c7851:87D'>
77        <UML:Multiplicity.range>
78         <UML:MultiplicityRange xmi.id = '127-0-1-1--172131be:13ab06c7851:87E'
79         lower = '1' upper = '1'/>
80        </UML:Multiplicity.range>
81       </UML:Multiplicity>
82      </UML:StructuralFeature.multiplicity>
83    </UML:Attribute>
84    <UML:Attribute xmi.id = '127-0-1-1--172131be:13ab06c7851:87F'
85      name = 'number' visibility = 'public' isSpecification = 'false'
86      changeability = 'changeable' targetScope = 'instance'>
```

Figure 2 : Sample code snippet from an XMI file of a class diagram

The xmi tag *<UML: Class>* is used to indicate the start of a class. One of the important attributes of this tag is the 'name' which indicates the class name. Under the *<UML: Class>* tag, it define the different attributes of this class by using the tag *<UML: Attribute>*. This tag also has some attributes like 'xmi.id', 'name', 'visibility', etc. given as arguments. The function of our *XMI-Parser* is to parse this xmi file and retrieve the values of attributes of the different tags. Also, from this class xmi file, we can find another important tag *<UML: Operation>* with an xmi id as same as that of the class in which this operation is defined. This tag has some important attributes like 'name' to indicate the operation or function name, 'visibility', etc. Figure 3 shows the details of this tag.

```
269 <UML:Operation xmi.id = '127-0-1-1--172131be:13ab06c7851:8B7'
270  name = 'confirm' visibility = 'public' ownerScope = 'instance'
271  isQuery = 'false' isRoot = 'false' isLeaf = 'false'
272  isAbstract = 'false'>
273  <UML:BehavioralFeature.parameter>
274   <UML:Parameter xmi.id = '127-0-1-1--172131be:13ab06c7851:8B8'
275    name = 'return' isSpecification = 'false' kind = 'return'/>
276  </UML:BehavioralFeature.parameter>
277 </UML:Operation>
278
279 <UML:Operation xmi.id = '127-0-1-1--172131be:13ab06c7851:8B9'
280  name = 'close' visibility = 'public' ownerScope = 'instance'
281  isQuery = 'false' isRoot = 'false' isLeaf = 'false'
282  isAbstract = 'false'>
283  <UML:BehavioralFeature.parameter>
284   <UML:Parameter xmi.id = '127-0-1-1--172131be:13ab06c7851:8BA'
285    name = 'return' isSpecification = 'false' kind = 'return'/>
286  </UML:BehavioralFeature.parameter>
287 </UML:Operation>
```

Figure 3: Sample code snippet for the *<UML: Operation>* tag

In the same manner, The *XMI-Parser* parses an xmi file for the object diagram and collects the state values of different attributes of different objects into permanent store. This permanent store will be populated later at the time of application execution. Figure 4 below shows a sample code snippet of an xmi file of an object diagram.

.

```
81 <UML:Class visibility="public" xmi.id="wAwFgXXpyxc7" name="Eraser:Item">
82  <UML:Classifier.feature>
83   <UML:Attribute visibility="private" xmi.id="PbvIAvbHidr5" initialValue="2" name="iid"/>
84   <UML:Attribute visibility="private" xmi.id="rvyXTVkKGDCe" initialValue="Eraser" name="iname"/>
85   <UML:Attribute visibility="private" xmi.id="xLOd0jVJ6pz0" initialValue="5" name="price"/>
86   <UML:Attribute visibility="private" xmi.id="OBUZNk4wf5QA" initialValue="1000" name="stock"/>
87  </UML:Classifier.feature>
88 </UML:Class>
```

Figure 4: Sample code snippet from an XMI file of an object diagram

The tag *<UML:Attribute>* define the attribute of an object. So our *XMI-Parser* check for these tags in the xmi file and records the different attributes of an object with their current state value into the Provenance Store. This state values will

be changed later at the time of application execution and are recorded into the Provenance Store with the help of an *Application Tracer*.

3.5 THE APPLICATION TRACER

The *Application Tracer* is responsible for capturing the provenance at the time of application execution. It provides a set of methods for the application developer so that the developer can directly include these methods in their application for capturing the provenance at the run time of the application. These *Application Tracer* methods are called from all the operations or function calls in the application which is already recorded in the *Opstore*.

The *Application Tracer* here acts as an interface for querying the provenance information from the permanent store. It is the responsibility of the application developer to develop an *Application Tracer* as part of the application development. Here the *Application Tracer* traces the attribute state changes of each object in the application and records the corresponding changes in the permanent store.

4. Conclusions

In this paper, we have described a scheme for creating provenance-aware application by using the basic UML design diagrams. We have described the scheme by three distinct phases *(i)* design analysis phase, where the design diagrams are analysed for capturing the initial process documentation details *(ii)* system initialisation phase, where it uses three permanent stores for initialising the process of capturing the provenance *(iii)* provenance collection phase, in which the provenance information are captured at run time of the application with the help of an *Application Tracer*. In this paper we have only considered class and object diagram. The class and object diagram gives us static information. By including interaction diagram the dynamic behaviour of the application can be studied. The information thus provided can help in deciding more accurate application tracer adaptation points.

REFERENCES

[1] Paul Groth, Sheng Jiang, Simon Miles, Steve Munroe, Victor Tan, Soa Tsasakou, and Luc Moreau. An Architecture for provenance systems. Technical report, University of Southampton, February 2006.

[2] Amin Vahdat and Thomas Anderson. Transparent result caching. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '98, pages 3-3, Berkeley, CA, USA, 1998. USENIX Association.

[3] Jun Zhao, Carole Goble, Mark Greenwood, Chris Wroe, and Robert Stevens. Annotating, linking, and browsing provenance logs for e-science. In *In. Proc. of the Workshop on Semantic Web Technologies for Searching and Retrieving Scientific Data*, pages 158 - 176, 2003.

[4] D. P. Lanter. *Lineage in GIS: The Problem and a Solution*. Technical paper. National Center for Geographic Information and Analysis, 1990.

[5] I. Foster, J. Vockler, M. Wilde, and Yong Zhao. Chimera: a virtual data system for representing, querying, and automating data derivation. In *Scientific and Statistical Database Management, 2002. Proceedings. 14th International Conference on*, pages 37 - 46, 2002.

[6] Peter Buneman, Sanjeev Khanna and Wang-chiew Tan. Why and Where: A Characterization of Data Provenance. In *In ICDT*, pages 316 – 330. Springer, 2001.

[7] Laura Chiticariu, Wang-Chiew Tan, and Gaurav Vijayvargiya. DBNotes: a post-it system for relational databases based on provenance. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD ’05, pages 942 – 944, Ney York, NY, USA, 2005. ACM.

[8] Simon Miles, Paul Groth, Steve Munroe, and Luc Moreau. Prime: A Methodology for Developing Provenance-aware Applications. *ACM Trans. Softw. Eng. Methodol.*, 20(3):8:1 – 8: 42, August 2011.