Supporting Scientific Collaboration Through Class-Based Object Versioning

Johnson Mwebaze[1,2], Danny Boxhoorn[2], and Edwin Valentijn[2]

[1] Makerere University, P.O. Box 7062, Kampala, Uganda
[2] University of Groningen, Landleven 12, 9700 AV Groningen, The Netherlands,

Abstract. Reuse of scientific data is central to much of science. Although data produced by individual researchers and groups is made publicly available, effective sharing is often prevented by lack of common resource discovery mechanisms and by format interoperability issues. Unlike commercial databases that operate fixed programmes (e.g. mortgage plan) and variable data (e.g. interest), in a scientific environment the reverse applies and the methods to process the data changes while the original data items themselves stay unchanged. Scientists often build on existing work and try different techniques for processing datasets, necessitating changing methods. In this paper, we provide a class-based object versioning framework that supports dynamic changes to pipelines while managing dependencies. The framework addresses the management of arbitrary changes made to scripts during a data flow and the association of these changes to data created.

Keywords: data lineage, provenance, scientific computing, code-sharing, data reuse

1 Introduction

During a scientific data analysis work cycle not all intermediate or working versions/variations of scripts become part of the software repository and therefore not all versions/variations of classes during the scientific analysis process will necessarily be released and made part of the software repository. The way people use versioning systems is by committing changes from time to time when they feel that they have completed a feature, but what happens between two commits is a mystery. For example, a user could make a change to a script, process and store a result A, makes another change, processes and stores the result A'. Both A and A' are committed into a centralized database but we do not know the difference between A and A'. The user could possibly commit his code after making and processing several other results. We can only make some hypothesis about the difference between A and A' by considering the two successive states of the script at hand, which could lead to a false negatives.

A scientist faces two principal obstacles when working with data: firstly, understanding the origins of data (i.e., *data provenance* [4], and secondly, understanding the differences between two data items (i.e., *object versioning*). Provenance systems [4] [6] do trace lineage by capturing and storing a complete trace

of a data flow. However changes made to the scripts are usually ignored. At a coarse level, provenance systems refer to `'a procedure x'` was run on `'data y'` [7] but at the lowest, scientists are also interested in knowing what changes in `'procedure x'` made data to appear the way it is. Capturing provenance is also possible when all users sign up to a systematic approach of processing and storage. However this is not often the case. For example, `Astro-WISE` (`AWE`) [2] provides scientists with an `AWE` environment and allows users to have their own code. Users can modify code in their repository to evaluate their specific questions about the datasets, change/apply their own algorithms on datasets and derive results following their insights. For each object processed using `AWE` we are able to keep truck of all dependencies, and processing parameters. However, tracing what users are doing with the code in their own repositories and how this code is affecting published data is still a challenge. Knowing the changes done to script and the data created, is of utmost importance especially in scientific collaboration which allows scientists to collectively reuse data, modify and adapt scripts developed by their peers to process data while publishing the results to a centralized data store.

Most scientific environments depend on the versioning capabilities available in versioning systems (e.g.CVS, SVN)to keep a log on how SC has changed over time. However, these tools are limited in their ability to detect differences in programs because they provide purely textual differences [8]. The level granularity of all of the commonly used versioning systems are file and/or program based and to some extent lines. While files/programs are too coarse-grained for detailed analysis, lines seem to be too fine-grained. The versioning information considered by these tools is the number of files, directories (and other dependencies) and their relationships, as well as lines added, deleted or modified for each commit(version). This might be helpful to a software developer, but not a scientist who is trying to understand the differences in his data to establish relevant links between data and changes in the program elements.

The Class-Based Object Versioning (COVA) we propose here will version data/source code (SC), in much finer-grained ways. The versioning will be based on program entities (like classes, methods, attributes) and class hierarchies relationships while linking this information to the data objects. This information will be made persistent in the database which can then be queried directly, without needing costly parsing steps. Scientists can then focus on more precise relationships and exploit the relationships to evaluate the variations of the lines of code (e.g., of a single method) and how the variation affects the data. We would like at the lowest level to link the changes made to program entities that do change results of a class to the data objects created. If two data objects were created by different variations (versions) of the same class, the difference between the two objects can be explained.

The rest of the paper is organized as follows; We present the underlying design objectives of COVA in Section 2. In Section 3, we show the relevance of this work to Linked Science. In Section 4 we present the framework for change detection. Object linking and version management is presented in Section 5.

We evaluate this work in Section 6. We review related work and conclude in Section 7.

2 Class-Based Object Versioning (COVA)

We want to enable scientists to iterate quickly on data processing and analysis tasks while associating changes made to SC to the data created. To do so, we include a COVA mechanism in `AWE` environment to automatically detect and analyze SC and manage dependencies between code edits and saved results. A code edit can be lexical, syntactical or semantic. For this work we only consider changes that would change the results of a class. The requirement is that, given a class with same input data and arguments, the class should always return the same derived object. Source edits that change spacing, comments, and other minor cosmetic tweaks that do not alter a function's behavior are ignored. This technique works as follows:

The scientist runs a script in `AWE` environment. If this is the first time this script has been run, a request to save the computed results to the database, will automatically create a code object. A code object(CO) is persistent object that stores information extracted from SC. e.g., classes, methods, class attributed and variables, class dependency relationships, in a database while linking the complete SC file with this information. The persistent CO will have a version number which is used to reference the CO to the derived object.

During subsequent runs of the same script (possibly after some edits), `AWE` compares the current script with the SC file linked to the persistent CO to detect changes. This comparison is only done when a user requests to commit the results.

If changes are detected `AWE` automatically creates a new version of the CO and creates dependencies between the CO, the created data objects and the SC edits.

2.1 Version Creation and Versioning Units (VUs)

An important aspect of the COVA is how to select the elements that shall be compared. These elements are the VUs that form the basis of comparison. We define a VU as an element associated to versioning information. A new version of the element is created when any part of it is modified.

For object-oriented systems, the state of an object depends not only on its own data attributes but also on the objects it refers to. A class that may use services of other classes to create an object. In such cases, a class will have elements to be versioned which are distributed through different files. A class-hierarchy graph is a straight forward relationship between classes. Class-hierarchy changes may affect calls to methods in any classes in the hierarchy. A straight forward approach for selecting VUs, is to consider all program entities in a class-hierarchy. For example, a class may be considered as an atomic version unit and also its methods, and attributes as other VUs. A change in any of the VU would create a new version of a class or a new version of a related class that uses the changed VU.

A version attached to an object must provide a snapshot of the VUs at the time of processing. Each VU may have its own version but the aggregation of these VUs (the class) may have another version i.e., a version number attached to an object should have the state of all classes, methods and attributes that were used at the time of making the object. For example, a `Python` class is composed of methods and attributes (locally defined in the class and inherited). In this scenario, a class is as VU and any of its methods and attributes are as also VUs. If one attribute/method or an inherited class is changed, a new version of the class is also created, because the class has been indirectly changed. Therefore the object processed with this changed class will have a new version, even when the most specific class may not have changed.

This versioning scheme we propose allows future queries over a specific versions of an object, which will provide a complete reconstruction of the all the classes that were used to derive the specific version of the object. There are various kinds of versioning problems in software, all of which pertain to compatibility between objects created and the classes used to access the objects. Differing versions of the classes may or may not be able to handle each others' data storage formats. A class's methods and fields may not maintain the same meaning as the class evolves, this means existing programs may break in places where those methods and fields are used. Therefore programs that load executable code at runtime must be able to identify the correct version of the class. Serialized `Python` object streams do not contain bytecodes, they contain the information necessary to reconstruct an object assuming you have the class files available to build the object. This means that all objects, class/module dependencies must all be coordinated to ensure that the executables are built from the correct versions of the source files.

3 Linked Data, Code and Results(Science)

This work exploits object linking to enable discovery of data by following links between data items, facilitate code/data reuse, and in effect reduce redundancy by reprocessing only data products that have not been processed before.

Quality control is typically one of the challenges in the chain of processing from raw data of the "sensor networks" to scientific papers. It requires an environment in which all non-manual qualification is automated and the scientist can graphically inspect where needed by easily going back and forth through the data and metadata of the whole processing chain for large numbers of data products. At the same time, the human and financial resources often dictate that not only the large survey teams are spread over many institutes in many countries, but also the data storage and parallel computing resources. This brings us well beyond the era of science on a desktop and into a paradigm in which astronomers, calibration scientists and computer scientists spread over a dozen or more locations in many countries link and share their work, SC and latest results in a environment that allows the quality control, re-processing and data discovery. This is demonstrated by the creation of the Virtual Observatory(VO)*

* http://www.euro-vo.org/

by the astronomers. The concept is that astronomers should have access to all the world's astronomical data, results and can link published work to the data. The VO is developing standards for linking various archives and data centers together. This linked data shall be accessed the same way people have access to any datum by linking his/her computer to the web.

Although the astronomers have not deployed the Resource Description Framework (RDF) and the Hypertext Transfer Protocol (HTTP) to publish structured data on the Web and to connect data between different data sources, they have used the VO to conceptually address the same problem (linking archives, data, and results). However since the Web is increasingly understood as a global information space consisting not just of linked documents and Linked Data, it would be interesting to investigate how Linked data framework can be applied to the VO.

The data of the very large majorities of surveys is fully public. Any astronomer is entitled to a copy of the data. Survey data is often used for new science cases the original designers of the survey were not planning to do themselves or did not foresee. In fact, many of the surveys are designed with the intention of precisely this happening. To be able to do this successfully requires that everyone is provided access to detailed information on the existing calibrational procedures and resulting quality of the data at every stage of the processing, that is, have access to the data and the metadata including process configuration at every step in the chain from raw data to final data products and most importantly detailed SC that was used to process the data. Sometimes, the unanticipated use-case specific demands may require re-processing of data sometimes starting from the raw data or probably requiring modifications in SC taking advantage of already existing work and an improved understanding/insights into the computational methods.

Another requirement with the physicists is the long-term preservation of the data, and processed results and the ability of recalibrating (re-processing) it to the requirements of new science cases. With time, software that is used to process data evolves with no backward compatibility support, this is where the linking of SC to the data (i.e., COVA) is very also important. With virtually many achieves linked together, finding of relevant datasets becomes another problem. This work includes another aspect of locating data of interest by combing code-based searches with provenance. This framework represents SC is as objects. Rather than viewing SC as linear streams of ASCII characters, we can now view SC as objects (COs). It will certainly allow extreme data validation, data reuse, ability to view data in many ways, and to search for data or code by any attribute/key.

4 Change Detection

To determine if a new version of a class has to be created, two classes are matched together to find the differences. We use two metrics to determine a matching between two classes. The first metric is the semantic difference in the class and the second metric is if the changes identified affect the results of a computation.

For each class, we generate and build a dependency graph. We denote a dependency graph of a class as node-labeled directed graph $G(V, E)$ where $V(G)$ denotes the set of all nodes in G and $E(G)$ denote the set of all edges in G.. The graph contains class nodes (non-leaf) and methods (leaf) nodes for each method in a class. For each method node, there is an edge connecting the method node to the class node. For a derived class, there exists an edge between the class and the classes from which it inherits. This process of adding nodes is recursive for the depth and with of the inheritance hierarchies.

The dependency graph represents the class and its interaction with other classes. This graph accounts for effects of inheritance, scoping, polymorphism and dynamic binding. What remains is to find the difference between the graph generated for the original class (G) and the graph of the modified class (G').

4.1 Class Matching

For each dependency graph of a class, we begin by matching the classes (non-leaf nodes), then for each class match we match the methods and attributes (leaf) nodes. To compare method nodes, we do not build an enhanced control-flow graph as work done in [1], we instead compare `Python` bytecodes. We do so because we want to avoid source edits that change spacing, comments, and other minor cosmetic tweaks do not alter a function's behavior. If changes are detected during method comparison, we continue to check if the results of the two methods differ. i.e., given two methods M and M' and same input dataset o, if $M(o) \neq M'(o)$, then we assume the changes made to M to create M', are significant to create a new version of a method, and eventually a new version of a class. The actual differences between the two methods are determined and logged.

We therefore provide a new graph representation and a differencing algorithm that will identify and classify changes between two graphs corresponding to a class while comparing known results of changed methods to verify the effect of the changes on the methods. We then associate the detected changes between classes and/or methods to the derived data objects that shall be created through the modified classes.

Node Matching To find differences between two graphs, we carry out a matching between corresponding nodes. Class nodes are matched to class nodes and method nodes are matched to methods nodes. For two graphs G and G'. Non-Leaf nodes that appear in G that do not appear in G' are deleted classes, whereas non-leaf nodes in G' and not in G are added classes. The same applies to leaf nodes. However if a class overrides a method in one class, the method will appear as a new method in overriding class. To ensure uniqueness of a node, we introduce the *pathToNode* for this purpose which is defined as follows;

Let V denote a set of all nodes in Tree (T), if $v \in V$, the $pathToNode(v_n) = v_1.v_2 \ldots v_{n-1}.v_n$, where v_1 is the root of T, and $v_1, \ldots v_{n_1}, v_n$ is the path from root v_1 to v_n. The '.' represents a link property attribute which defines relationships between two nodes that are transitively connected.

Based on the definition of the *pathToNode*, we define a matching M as below;

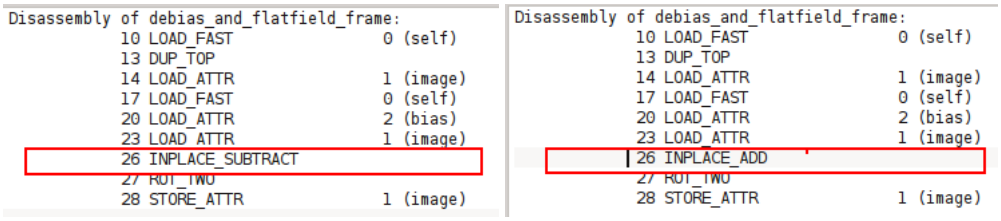
Given set of node pairs (v_a, v_b) where $v_a \in V_a$ and $v_b \in V_b$, M is called a matching from T_a to T_b, *iff*

1. $v_a, v_b \in M$, $v_a \in V_a$, $v_b \in V_b$, pathToNode(v_a) = pathToNode(v_b)
2. $\forall (v_{a1}, v_{b1}) \in M$ and $(v_{a2}, v_{b2}) \in M$, $v_{a1} = v_{a2}$ *iff* $v_{b1} = v_{b2}$
3. Given $(v_a, v_b) \in M$, suppose v'_a is a parent of v_a and v'_b is the parent of v_b, then $(v'_a, v'_b) \in M$
4. bytecode(v_a) == bytecode(v_b)

We now use the *pathToNode* and matching definitions to recursively match nodes in G and G'. We begin the comparison at the class level. After matching classes we then match methods for each pair of unmatched classes. Unmatched classes are those classes that have differences in their implementation. For any unmatched methods, we compare the output of two methods and we continue to log the semantic differences if the output of the methods differ. This process is summarized in algorithm 4.1

```
Input: Original Class C, Modified class C'
Output: set of unmatched methods classes, C'' unmatched methods M'', set of semantic differences DIFF
  Parse classes C and C' into their dependency graphs G and G' respectively
  T ← V(G) Set of all none leaf nodes of G
  T' ← V(G') Set of all none leaf nodes of G'
  for each node in t ∈ T and t' ∈ T' do
    matched ← match(t, t')
    if matched then
      t, t' are equivalent
      continue
    else
      V ← V(t) Set of all leaf nodes of t
      V' ← V(t') Set of all leaf nodes of t'
      for every node m in V do
        for every node m' in V' do
          matched ← match(m, m')
          if not matched then
            compare the output of both methods given the same input data
            if output differ then
              add m, m' to M'', remove m and m' from V and V' respectively
              add semantic differences between m and m' to DIFF
            else
              remove m and m' from V and V' respectively
          else
            remove m and m' from V and V' respectively
      if M'' is not empty, add t, t' to C''
      remaining in V and V' are new nodes(methods)
```

Algorithm 4.1: Change Detection



(a) (b)

Fig. 1: Disassembled bytecode of `debias_and_flatfield_frame`

Comparing Output of two methods Given the same list of source code segments, processing environment and arguments, a compilation should always return the same derived object. If the results defer, then we can confirm a change in the implementation. We note that when external imports e.g., numpy, pyfits are modified, these too can change a result of a method. To diminish the effects of such changes, the test environment includes a standard setup with known versions for external exports and expected output from each method. The modified methods are plugged into this test environment while other dependencies remain constant. We only execute sections that have been modified.

If a method M was modified to M', we check to see if for any object o, $o.M() == o.M'()$. If N, and M are two methods, where N precedes M during execution, if the state of object o, at the time the compiler completes the execution of N is o' i.e., $o.N() = o'$. and the $o'.M() = o''$ then if $o'.M' \neq o''$ then we can confirm that changes in M, are significant to create a new version of M i.e., M'.

Semantic differences between methods To get the semantic differences between methods that have not been matched, we use the `Python dis` module to disassemble the bytecodes of the unmatched methods. Fig. 1 shows the disassembly of two methods. Fig. 1(a) is the original method while Fig. 1(b) is the modified method. The changed lines are highlighted in red. In the method `debias_and_flatfieldframe` the operation between attributes `self.image` and `self.bias.image` was `INPLACE_SUBTRACT`, that has been changed to `INPLACE_ADD` in the modified method.

5 Object Linking and Version Management

We describe in this section `AWE`'s mechanisms for linking objects and how we manage versions between objects. We have used persistent COs that represent a set of closely related parameters extracted from a class that are represented as an object stored in a relational database.

5.1 Persistent and Code Objects

The persistent object hierarchy makes the core of this framework. We automatically make all objects persistent in the database as attributes, as fully integrated objects or as descriptors. Source code files are not stored in the database, however their unique filenames and links to their COs are stored in the database.

Each versioned class has an associated CO from which the versions of the derived objects are obtained. Each CO knows all its dependencies (methods and relationships with other COs) and their states (versions). Each CO is linked to other COs which can themselves be linked to other COs. These COs constitute a class-hierarchy relationship. The highest CO in the hierarchy being the most specific class for the derived object. The dependency hierarchy graph of a CO can be represented as version graph. Each version graph will be same for all derived objects of same version.

For each data object processed, a persistent link to the version of CO that was used to make the object is created and is stored as the part of the object's attributes (or metadata). This ensures that during the object's de-serialization the appropriate classes are called to reconstruct the object. The CO is identified by a unique object identifier (*object_id*) and version number *version_no*. The *object_id* and the *version_no* of a CO are used as reference to identify the relationship between the SC and the derived object. A class (or derived object) is linked to the CO through a persistent attribute called `code_object`.

Definition of persistent attributes. All `AWE` classes are derived from a customized metaclass. Using the metaclass we can then manipulate class creation, object instantiation and method execution. We defined another class `DBObject` which is the root class of the hierarchy of the persistent classes. This class defines the primary key `object_id` of all objects. `DBObject` is derived from the metaclass. Any classes that inherits the `DBObject`, automatically becomes persistent. This creates all the necessary schema structures, such that attributes and data created or used during the processing will be stored. Likewise, a persistent attribute is defined by using the following expression in the class definition.

```
attribute_name = persistent('A doc-string for the attribute',
                 attribute_type, attribute_default)
```

If attribute_type is a subclass of `DBObject`, then the attribute will be a link, else the attribute will be a descriptor. If the attribute_default is a list, then the attribute will be an array of objects of that type. If persistent has only a string argument, then the attribute is assumed to be a link to an object of the same type as the class it is defined in. We present an example below

```
class ClassB(DBObject):
    e = persistent('', ClassA, (ClassA, (), {})
    f = persistent('', ClassA, [])
    g = persistent('')
    filename = persistent('File part', str, '')
class ClassC(ClassB):
    h = persistent('', ClassD, None)
```

ClassB defines four links: 'e' is a link to an instance of ClassA, 'f' is a array of links to instances of A (default empty), 'g' is a link to another instance of ClassB and filename attribute supports the storage of files. Note that persistent properties are inherited. So ClassC defines a new persistent object, with four persistent attributes ('e', 'f', 'g', 'h').

5.2 Version Control

Our requirement is that we are able to test the equivalence of two COs. Each CO has a unique identity, a version attribute and a version predecessor attribute. If a CO is created as a new version of another existing CO, its version predecessor pointer points to its version predecessor otherwise its version predecessor pointer is null. Each CO is read-only. Changes made to an existing CO are stored as new version of the CO.

A new version will be created for all unmatched classes and unmatched methods. We have created versioned groups we have called 'type' which are identified by the class names or method name. For each type, the version number counter begins from 1 and incremented by 1. For example, the type of a class called `BaseFrame` will be BaseFrame. Three basic operations for version control are new, edit and delete. Each of these operations is modeled as a function that takes a class as its major input and returns a new version and a CO stored in the database.

The `New` operation creates and adds new CO to a database. The CO represents new type which is different from any existing type in the database. This new CO will have no version predecessor. The `Edit` operation is used to create a new (edited) version of an existing type in the database. This operation takes as input a class that is assumed to be modified and matches the class against an existing CO. Based on the results of matching a new version of a CO might be created. If a new version of the CO is created, the version predecessor attribute of the new version will point to the CO that was used during the matching. A user has an option of selecting a specific version of the CO to use during matching, otherwise the latest version will be used during the matching process. The `Delete` operation is used to remove a version represented by a CO from the database. If the CO being removed is referred to by other CO then the Delete operation will not be successful.

6 Evaluation

We present in this section a few implementation details for the system in relation to processing data objects and querying COs.

The `AWEPIPE` environment variable specifies to `AWE` where the local personalized checkout is stored. It is the classes in the personalized checkout that are used to process data. After the processing of a target, the class that was used to make the target(data object) are matched against the CO of the same type stored as the database.

To process a target, a researcher begins by sending a query to the database for the target. If the target exists, the target is returned, else the processing of the target is initiated. The inputs to the pipeline are either queried from the database if they already exist or created on the fly if the objects are not *uptodate* or do not exist. This is a recursive process that begins from the target to the raw data as observed from the telescope. An object is *uptodate* if the most specific class and its dependencies that were used to make the object have not changed or if any of its dependencies have not been made by a newer version of a class. This

is determined by comparing version of CO that was used to make the object and version of the current (latest) CO. `AWE` provides a web-service for this check, an example of this comparison is shown in Fig. 2. If a newer version of classes exists it highlighted in orange. The object viewer of each object would give details of the changed attributes.

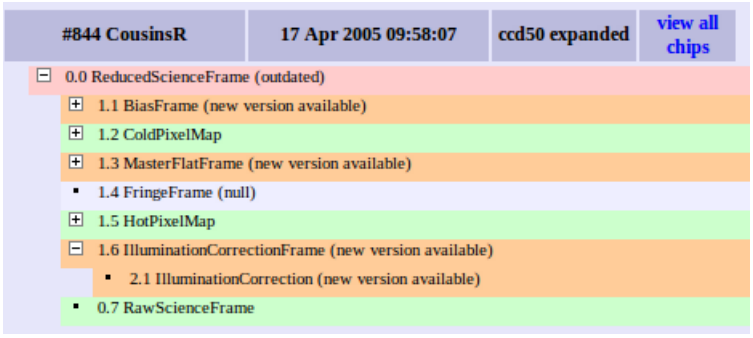


Fig. 2: Dependency graph of classes required to make the `ReducedScienceFrame` object

Querying Versioning Information. We have defined a notation that is based on the idea that a class is in some sense equivalent to the set of all its instances. To illustrate the concept, let us give a few examples. Given a persistent class X with persistent property y, then the expression $X.y == 5$ represents the set of all instances x of X, or subclasses of X, for which $x.y == 5$ is true. To obtain these objects the expression needs to be evaluated, which can be done by passing it to the select function, which returns a list of objects satisfying the selection. `AWE` publishes data to the virtual observatory. Objects from `AWE` can be displayed with all its data and code provenance information.

7 Related Work and Conclusion

There are a number of existing techniques for computing differences between two versions that also recognize differences in object-oriented features. Semantic diff [5], compares two versions of a program procedure-by-procedure. However, it does not consider dependencies relationship between classes and variables. BMAT [9] performs matching on both code and data blocks between two versions of a program in binary format, however it does not provide information about differences between matched entities. JDIFF [1], uses the OOP approach when comparing classes. Its main focus is to determine differences that change the behavior of a program, e.g., changing branch conditions. Our focus is not behavior of the program, but changes that have an effect on the results of a program. There is also considerable work related to Version Control Systems (VCSs) [8], however their focus is dedicated on modeling software artifacts and

therefore version numbers created by VCSs are opaque identifiers and as such can not be used for object versioning.

We could leverage on some of the change detection tools, however, all users have to sign up to a systematic approach of processing and storage. Specifications for research code (as compared to production-quality code) are often ill-defined and constantly-changing likewise a typical data analysis work cycle is a recursive process where scientists change methods (or an implementation) or may even disagree on some implementations. In a such an environment, not all these changes would become part of the SC repository.

The work done in this paper, allows such kind of changes to be stored while linking them to the objects they created. All variations of implementations created during the scientific processes become publicly available and can be used to reprocess data, to find data through code-based searches and to understand the process the lead to the creation of a data item. This framework allows portability of data and reuse since each derived objects knows specifically how it was made.

References

1. Apiwattanapong, T., Orso, A., Harrold, M.J.: A differencing algorithm for object-oriented programs. In: Proceedings of the 19th IEEE international conference on Automated software engineering. pp. 2–13. IEEE Computer Society, Washington, DC, USA (2004)
2. Begeman, K.G., Belikov, A.N., Boxhoorn, D.R., Dijkstra, F., Valentijn, E.A., Vriend, W.J., Zhao, Z.: Merging grid technologies. Journal of Grid Computing 8, 199– 221 (2010)
3. Conte, D., Foggia, P., Sansone, C., Vento, M.: Thirty years of graph matching in pattern recognition. IJPRAI pp. 265–298 (2004)
4. Freire, J., Koop, D., Santos, E., Silva, C.T.: Provenance for computational tasks: A survey. Computing in Science and Engineering 10(3), 11–21 (2008)
5. Jackson, D., Ladd, D.A.: Semantic diff: A tool for summarizing the effects of modifications. In: Proceedings of the International Conference on Software Maintenance. pp. 243–252. ICSM '94, IEEE Computer Society, Washington, DC, USA (1994)
6. Moreau, L.: The foundations for provenance on the web. Found. Trends Web Sci. 2, 99–241 (February 2010), `http://dx.doi.org/10.1561/1800000010`
7. Ogasawara, E., Rangel, P., Murta, L., Werner, C., Mattoso, M.: Comparison and versioning of scientific workflows. In: Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models. pp. 25–30. CVSM '09, IEEE Computer Society, Washington, DC, USA (2009)
8. Oliveira, H., Murta, L., Werner, C.: Odyssey-vcs: a flexible version control system for uml model elements. In: Proceedings of the 12th international workshop on Software configuration management. pp. 1–16. SCM '05, ACM, New York, NY, USA (2005)
9. Wang, Z., Pierce, K., McFarling, S.: Bmat - a binary matching tool for stale profile propagation. The Journal of Instruction-Level Parallelism 2, 2000 (2002)