

# Provenance in dynamically adjusted and partitioned workflows

# Provenance in Dynamically Adjusted and Partitioned Workflows

Daniel Goodman
Oxford e-Research Centre
7 Keble Road
Oxford, OX1 3QG
Daniel.Goodman@oerc.ox.ac.uk

***Abstract*—In this paper we describe the provenance system built into the distributed Martlet middleware. Due to both the need for scientific reproducibility, and to determine exactly what has happened with any given piece of analysis, it is necessary for this middleware to record detailed and structured provenance data in an easily query-able form. This is achieved through the use of integer clocks and directed graphs. Using these, this system is capable of keeping a complete history of the creation of all data, including the ability to store in-depth information defined by the task about the operations performed. This allows the system to continue to gather provenance data regardless of the rough grained functions being wrapped by the middleware.**

**The middleware was developed to support functions described in "Martlet", a workflow language developed to address the problem of how to analyse the data generated by the climate*prediction*.net experiment. This data is both highly distributed, and resides in a dynamic environment where the partitioning of data structures across the distributed nodes may change both in the number of pieces and their locations, and resources may come and go. This makes it necessary for the structure of the workflows to change from execution to execution. As such the provenance system is also required to be able to handle such a dynamic environment.**

## I. Introduction

In this paper we describe a Directed Acyclic Graph (DAG) based design and implementation of the provenance system integrated into the middleware [1] supporting Martlet [2], [3] workflows. This system is able to keep track of the events leading to the current state of every item of data within the system, including process defined information about each task performed. This data is structured such that it can be queried and filtered by the user with a range of tools, as well as providing scope for the user to extend this with custom constructed visitors [4] to explore the data. This is achieved through the use of DAGs and integer clocks. These DAGs are grown inductively throughout the life time of the different data structures.

Provenance [5] is of particular importance to the analysis done using the Martlet middleware because in addition to the standard requirement in scientific work for results to be reproducible, Martlet workflow execution results in not only the location of the computation and the data varying, but also the absolute structure of the functions being used to perform the analysis. This is done to overcome the changing partitioning and high distributed nature of the output from projects such as climate*prediction*.net [6], [7]. As these details are ordinarily hidden from the user, it is necessary to have a detailed provenance system if the user is to be able to ascertain exactly what computation was performed. This need is also furthered in a Grid environment where it is possible to publish URIs of data structures online for others to use. When this usage pattern occurs it is not necessarily sufficient for the user to record what he has done. Moreover, it is possible for such a system to record information about any operations performed on the data with greater detail, accuracy and fidelity than a user could manage.

There is already a huge volume of existing work on provenance systems [8], including much work on systems based on DAG's, and we could turn this paper into a survey of such work. However, the core difference between this work and the work that has gone before is that this work has to be able to handle the two tier data structures and the changing structures of the workflows from execution to execution provided by Martlet.

In this article we provide background by first briefly introducing the problem, and the language and middleware we have developed as a solution to this problem. We then look in detail at the implementation of the provenance system that is integrated into this middleware.

### A. Motivation for Martlet and its middleware

Martlet was developed to assist in the analysis of large quantities of dynamically distributed data generated by the climate*predication*.net (CPDN) [6], [7] experiment. Existing combinations of middleware and workflow languages are not sufficient for analysing the CPDN dataset, as while there is a wide range of different languages, compatible with different middleware, supporting different tools, databases, scientific equipment etc, all of them address the same programming model. In this model there are a known number of data inputs that need to be mapped to computational resources. As a result, none of them are able to handle data that is split into an unknown number of pieces at the time the workflow is submitted. This problem occurs because these workflow engines, although aware of the data type, do not *interpret* the data, but instead they simply pass it between functions. This restriction means that existing workflow engines are not able to take a generic workflow and use it to generate workflows that match the partitioning of data they are passed.

An example of a situation where the current solutions are ineffective is the calculation of an average across a list. Across a local list $[x_0, x_1, \ldots, x_{n-1}]$ a solution can be written as;

$$\overline{x} = \frac{\sum_{i=0}^{n-1} x_i}{n}$$

This is easily described with existing workflow languages. However if this list of numbers is now partitioned into $a$ pieces $[x_0, x_{(n_1-1)}], \ldots, [x_{n_{(a-1)}}, x_{(n_a-1)}]$, and the solution rewritten as:

$$y_0 = \sum_{i=0}^{n_1-1} x_i$$
$$z_0 = n_1$$

$$y_1 = \sum_{i=n_1}^{n_2-1} x_i$$
$$z_1 = n_2 - n_1$$
$$\vdots$$
$$y_{a-1} = \sum_{i=n_{a-1}}^{n_a-1} x_i$$
$$z_{a-1} = n_a - n_{a-1}$$

$$\overline{x} = \frac{\sum_{i=0}^{a-1} y_i}{\sum_{i=0}^{a-1} z_i}$$

This can now not be written in existing workflow languages unless the value of $a$ is known at the point the workflow is written, or the programmer includes the specifics of how to handle the variable number of arguments. This adds complexity to the system that the user does not want and may not be able to deal with, and simultaneously introduces a much greater potential for the insertion of errors into the process.

### B. Overview of Martlet

Martlet [2], [3] is the workflow language that the middleware containing this provenance system was designed to support. The language supports most of the common features expected of a workflow language, but it also has constructs inspired by the inductive constructs of functional programming languages [9], which it uses to abstract the parallelisation of the data and workflow. In addition it supports two classes of data structure, 'local' and 'distributed' to enable it to reason about the data it handles.

In functional programming languages it is possible to write extremely concise powerful functions based on recursion. For instance the reverse of a list of elements can be defined in Haskell as:

```
reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]
```

This simply states that, if the list is empty, the function should return an empty list, otherwise it should take the first element from the list and turn it into a singleton list. Then it should recursively call reverse on the rest of the list and concatenate the two lists back together. The importance of this example is the separation between the base case and the inductive case, which allows the function to avoid ever mentioning the length of the list. Using these ideas it has been possible to construct a language that abstracts the level of parallelisation of the data away from the user, leaving the user to define algorithms in terms of base cases and inductive cases. For example, the distributed average problem looked at in section I-A, taking the distributed dataset `a` and returning the average `b`, can be written in Martlet using a `map` statement to distribute the base case across an unknown number of pieces of partitioned data. These base cases generate for each partitioned piece of the dataset the sum and cardinality of these local sets. The distributed output from the base case results are then merged through addition to produce a pair of local results that represent the whole system. This is orchestrated through the use of a `tree` statement, taking the function to merge two results as an argument. As this is a function, the merge operation can be arbitrarily complex. The final result can then be calculated by a final division to produce the overall average for the distributed dataset.

The important thing to note about this function is that, although it works on partitioned data structures, at no point is a reference made to how the data structure is partitioned. The user therefore remains unaware of the partitioning and the function can be reused on differently partitioned datasets without having to make changes to the code.

### C. Middleware

The supporting middleware [1] is constructed using a service oriented architecture. This is built on top of Apache Axis [10] running on Jakarta Tomcat [11].

The middleware itself is designed using an architecture that is similar to MONET [12], taking on board ideas put forward by WS-GAF [13] and WS-RF [14] on the use of handles and URIs to manage resources. It is not however constructed with a strict adherence to any one set of ideas or principles.

The middleware is abstracted so the different conceptual parts of the system are separated into different concrete parts. The structure of the communication between these parts and the use of XML documents posted by each node on the web means that nodes can be added and removed at will, without having to inform the whole system of the change.

Data is passed by reference, which has two main advantages. First, the data is only moved when it is required, allowing the references to be passed through many services without having to move very large amounts of data through with them. Second, the use of references means that data transfer can be instigated by a more efficient method than SOAP [15]. As a result the data transfer model is abstracted so that the protocol used to transfer the data is separate from the implementation of the rest of the middleware, with lazy evaluation [16] being used to minimise the data transfer overhead.

As well as the use of references to pass data, references are used to pass functions. This allows functions to be first class values that can be passed into and used in other functions. This applies both to functions that have been submitted to the middleware using the Martlet language and to the base functions that are deployed to the system by administrators.

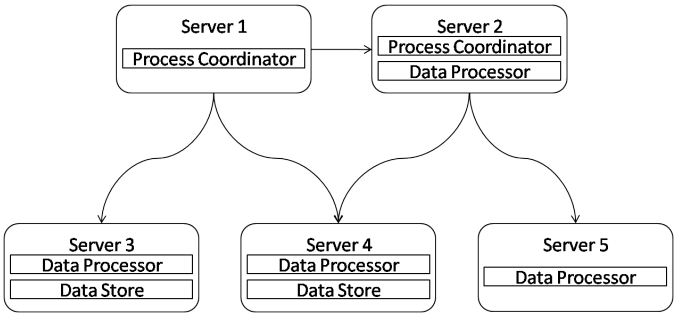



Fig. 1. An example of how five servers could be configured. Note that more than one Process Coordinator can use each Data Store and Data Processor and the Process Coordinator does not need to know about all available Data Processors.

The middleware also has a comprehensive locking and data transfer model that allows multiple functions to run at the same time, and functions to be queued without having to worry about forgetting data or generating race conditions. The locking system is extended to keep records, allowing the provenance system to be built into the middleware, keeping a complete record of the life of each piece of data.

*1) Topology:* The middleware is divided into three logical units: Data Stores, Data Processors and Process Coordinators.

*a) Data Stores:* provide a set of methods for accessing the data stored at a given location. This unit is deliberately lightweight and only capable of generating a data structure from stored data.

*b) Data Processors:* ingest, store and run Martlet abstract syntax trees on datasets, which they either have locally or retrieve from another Data Processor or a Data Store.

*c) Process Coordinators:* are the components that users interact with. They handle access to the rest of the middleware. This is consequently where most of the complexity of the project occurs, since this is where the generic trees that represent submitted functions are adjusted to fit the arguments on which the function has been called, and then are broken up and scheduled across the Data Processors. This is the only component to have any knowledge of other nodes in the system.

All three components share common functionality for transferring, discarding, and publishing data. They can be grouped together at will on servers, so it is possible to have both a Data Processor and a Data Store on the same machine. A possible configuration of five servers is shown in Figure 1. It is worth noting that many different Process Coordinators can use each Data Processor and Data Store concurrently and each Process Coordinator does not need to be aware of all the other nodes.

*2) Data Structures:* There are two classes of data structure supported by the middleware, 'local' and 'distributed'. Local data structures are any data structure that always resides on a single server. Distributed data structures are abstractly a list of local data structures. So for example a distributed matrix is a list of matrices, which, if concatenated together in the order they appear in the list, would produce a single matrix equivalent to the distributed matrix. This means that, when a new data structure is added, its distributed counterpart is automatically generated.

*3) Resource Lookup:* All data structures and functions are held in handles stored in a lookup table indexed by URIs. The URI's are constructed such that they are always unique to the server in question, so if data is copied between servers, a new URI will always be provided to maintain this uniqueness. The use of handles to hold the data structure or function facilitates the storage of meta-data, allowing it to be seamlessly passed around with the data structure or function. This meta-data consists of both object specific information and implementation specific information. Examples of this data include, the number of processes currently using a data structure, whether a data structure is locked to prevent race conditions, and when the data structure's existence can no longer be guaranteed.[1] This means that if a data structure is forgotten because of either the failure of a server or the forgetfulness of the user, the resources used to store it will be recovered automatically.

*4) Function Execution:* To allow both the use of legacy code and the concurrent operation of functions, the mechanism to call functions is broken into two pieces. Stored in the handle in the lookup table is a *Function Constructor*, this is stateless, and so thread safe. It has the task of taking a list of arguments and generating a *Function Object* from this list. The Function Object is an object which when invoked will perform a specific task on the arguments that where passed to the Function Constructor. As a unique Function Object is created for every function invocation, the complexity of concurrent calling is now removed. The Function Object itself is a wrapper for the code that is actually going to be called, providing a uniform interface for the rest of the middleware to interact with. This wrapper then in addition to providing a way of adding legacy code to the rest of the system handles the locking of data structures to prevent race conditions, and the operation of the provenance system.

## II. Provenance

The intrinsically dynamic and distributed nature of this programming model and architecture make the need for provenance even more pronounced than in other distributed applications. To address this, a provenance system built on distributed clocks and directed acyclic graphs has been constructed. Because the design and functionality of the middleware is detached from the functions being executed on the individual nodes, it is a necessary requirement of the provenance system to allow functions to add their own meta-data to the provenance system. It is also necessary for the system to be able to handle the two different classes of data structure and the hierarchy of functions that can exist in a manner such that the user is still able to sensibly query them.

[1]If there is space to continue storing a data structure there is no reason to actively destroy it. Instead this can be delayed until the garbage collector is invoked.

The DAG data structure in the provenance system, like the functions written in Martlet, is built up inductively over the life of all the individual data structures. The nodes on this graph can contain a range of different information, and, if required, these nodes could be added to or extended to gather other information that it is not possible or sensible to collect from the underlying tasks. Such information could include the overhead time taken to perform certain instructions within the middleware.

We will now look in more detail at the nodes, how this graph is created, and how it handles the complexity of the different classes of data structure; we will then examine how this data can then be queried and how this process might scale as the use of the middleware increases.

### A. Handles and Provenance Nodes

The handles used in the lookup table to store data structures and Function Constructors contain a range of meta-data. This meta-data includes for provenance, an integer clock that will record the largest value of any integer clock it encounters, and a reference pointing to the provenance DAG node that was created by the middleware the last time an operation occurred that wrote to the data structure. The nature of the provenance nodes themselves can vary according to the information that they need to store about how the data structure was last modified. However, they will all contain an integer recording the value of the integer clock when they were created.

Currently there are four ways that data structure can be created or modified, these are: by creating a new structure; by creating an empty handle to hold output from a function; by moving a data structure's location; and by executing a function that modifies the data structure.

### B. Data Structure Creation

The creation of data structures, or the empty handles that will hold them, is the base case for the construction of the provenance DAG. Because of two classes of data structure and the different types of data, there are multiple ways that this can occur, and these are introduced here.

*1) Local Data Structures:* The most basic type of data structure that maybe created is a Local data structure, either on a *Data Store* or on a *Data Processor*. These data structures are created by the user submitting a description of the structure they would like to create, either as part of a workflow, or as a Web Service call. These descriptions may be entirely self-contained, for example create an $n$ by $n$ identity matrix, or they may draw on information held on a Data Store, such as creating a matrix of climate model runs where each model produces a single column containing specified values, and each model satisfies a given set of conditions. With all of these cases the node in the provenance graph will contain at least the information used to create the data structure, and the integer recording the value of the integer clock in the handle at the time the node was created. As there are no vertices leaving this node, and at this stage none pointing to the node, it provides one of the base cases for the DAG.

*2) Distributed Data Structures:* Distributed data structures are created on *Process Coordinators* where if required, information about the state of the Data Stores is gained from a source, such as a database or a supporting Web Service. They then make the necessary requests to the Data Stores or Data Processors to construct the local data structures corresponding to the request. Once these have been created and their URI's have been returned, the list of URI's are placed in a handle, and the provenance reference set to reference a node that contains the data used to create the local data structures, or the data used to generate this query, and an integer recording the value of the handles clock at the time of creation. As with the local data structure there are no vertices currently pointing to this node, and none leaving it, so it is another base case.

*3) Empty Handles:* As functions require both the input and the output handles to be handed to them as arguments it is necessary to create empty handles for the output information to be placed in. These empty handles come in two forms, the simplest is for a local data structure, in which case a handle is produced containing no data structure. This handles provenance node just contains an integer storing the clock value of the handle.

When a distributed handle is required, it is necessary to determine how many pieces the data held by this handle will ultimately be distributed across, this information is gained by the user providing an argument containing a reference to an existing data structure of the correct partitioning. Once this is known, local empty handles for each of these can be created, and a distributed handle produced to hold the corresponding URI's. The provenance node in this distributed handle will then contain the information used to ascertain the distribution of the local handles, and a clock recording the value of the handles clock at the time of creation. Once more, as there are no vertices to or from these nodes at this time, these are the final examples of the possible base cases.

### C. Applying Functions

Having looked at the ways that the base cases of the provenance DAGs can be produced, we will now look at the process that takes these from being independent nodes to part of a larger DAG that grows as the users perform actions on the data structures, recording in depth information about the circumstances that lead to the current state of the data structure in the process.

The inductive step of the construction of these DAGs appears in several forms. The simplest of these is when a base function is called. A base function is a function that is not constructed by chaining together other functions supported by the middleware, but instead just calls an external function wrapped by the middleware in a Function Object. An example of such a function could be matrix multiplication provided in a numerical analysis library. When this is executed, the provenance system records this by updating all the argument handles integer clocks so that they now contain a value that is strictly larger than the largest value currently held by any of the argument handles. A new provenance node is then

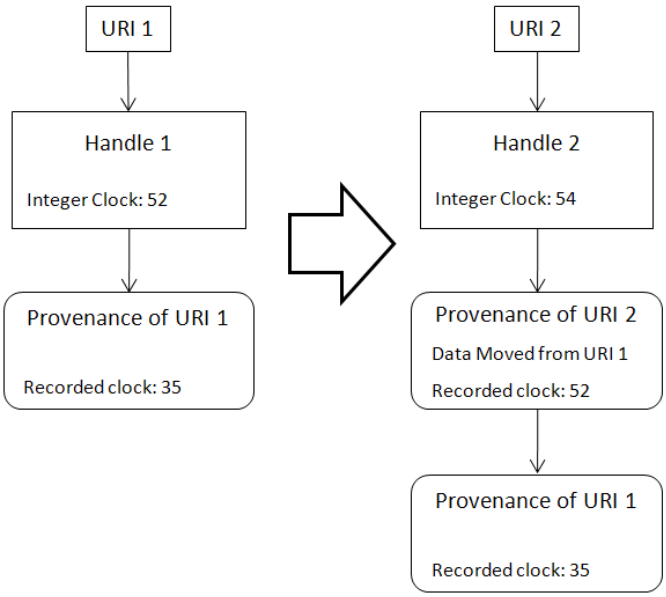



Fig. 2. A diagram illustrating the changes to the handles and the DAG when the data structure referenced by URI 1 is moved to a different server, and a new handle is created.

created containing the new value of the integer clocks, a list of vertices that reference the provenance nodes belonging to the arguments of the function, the URI of the function being executed, and finally a set of key value pairs. These key value pairs are the mechanism by which the middleware is able to store function specific data about the execution. This list of key value pairs can contain any information that the programmer of the wrapper wishes to store, and it is envisaged that they would be used to store information such as the completion state of the function, or elements of log files, however it is entirely up to the programmer to decide, and they could in principle contain a range of other information. Having constructed this node, all the handles that are marked as being written to by the executed function then have their reference to the provenance DAGs updated to reference the new provenance node. In doing so the graphs are now extended and joined to contain the history of how the data structures written to reached their new states. A diagram demonstrating this can be seen in Figure 3.

The next case is handling functions that are constructed through the composition of other existing functions in the middleware. In principle, it would be possible to do nothing for these cases and just allow the underlying calls to functions provide all the documentation. However in doing this a large amount of control flow information and structure would be lost, so instead, two nodes are used to provide the semantic equivalent of brackets around the function. The first of these is created and assigned to all of the arguments that will be written to, before the function is executed. The second node is added after the function has completed its execution. If we assume that there are no failures, all that would be required of each node is to have a list of vertices pointing to the nodes of the arguments, and one node to contain the URI of the function. However, since it is not realistic to make this assumption, both nodes contain the URI of the function. They also contain a unique reference, allowing them to be paired up and any abnormalities caused by a function failure to be detected. This can be seen in Figure 4.

The final case for the inductive steps is the separation between the DAGs being constructed within distributed data structures, and the DAGs created within the local data structures contained by the distributed structure. The strategy taken here is to just record the function calls on the distributed structures in the same way function calls are recorded for local structures. Then if more detail is required, the user can then retrieve the history of each individual piece of data. This separation means that the provenance systems model will remain intact, if, as planned, the Process Coordinator is placed onto other middleware from other projects, for example by using JIT compilers to produce output in other workflow languages. In addition, it is felt that this is not a major restriction as the focus of this project is to hide the distribution of the data. Where it is necessary to look in enough detail at results to determine the partitioning of the data that they were generated from, these results will normally be local data structures, so the partitioning information is already described in their own provenance DAG. If, in the future, it is decided that this separation is not appropriate, the extra information that needs to be added to encapsulate the functions can be achieved in the same way that hierarchies of functions are handled using nodes to encapsulate specific pieces of the DAG.

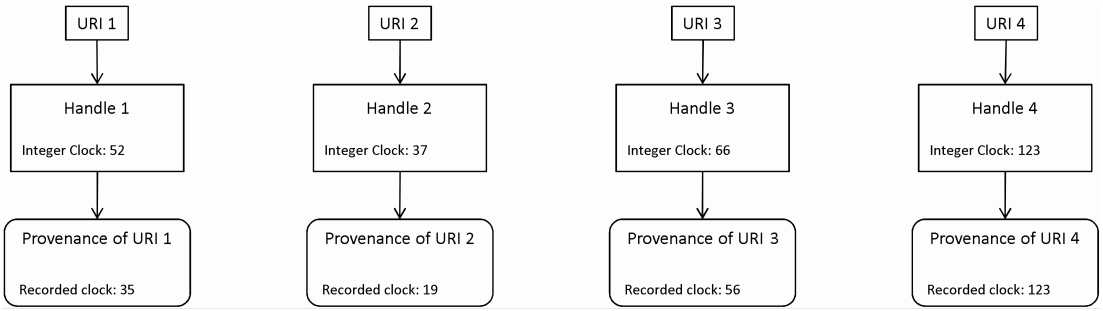
### D. Movement of Data

When a data structure is moved to a different server as part of a computation, the DAG that is reachable from the handle of the data structure is transferred with it. This ensures that, at any time, each data structure has access to the complete DAG data structure describing its history. There are practical issues relating to the relative sizes of the provenance data structures, and it is only practical to do this for applications with a sufficiently independent and rough grained usage of data structures. However this can be trivially overcome in a range of ways discussed in Section II-F.

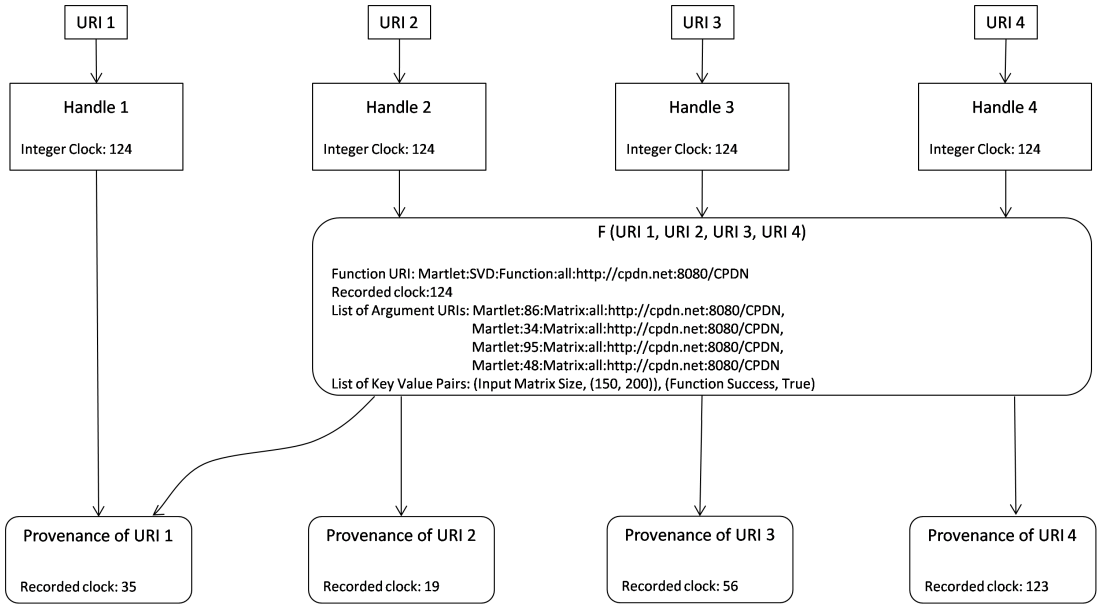
To allow the URI to carry on referencing the server containing the data structure, when a data structure is moved between servers, its URI is changed to reflect its new location. This change is recorded in the DAG through the addition of a new node at the root of the DAG documenting the change. The change of URI is important for the provenance because it ensures that all URI's only appear on a single server. This means that a URI combined with the clock value stored in a node is always unique to a node representing a specific event in the history of the data structure reference by that URI. A demonstration of this can be seen in Figure 2.

### E. Querying the Provenance Data

Having generated all this data about the sequence of events that have resulted in a given data structure reaching its current




3a: The initial state of the handles, with DAGs that can be assumed independent.




3b: The state of these handles and DAGs after F has been applied to the 4 arguments, URI's 2-4 have been written to.

Fig. 3. Diagrams representing the before and after states of the application of F to 4 arguments.

state, it is important to be able to query this information. The simplest means of performing this is to have a visitor [4] explore the DAG of a given data structure and produce a report on the sequence of events that resulted in the existence of this structure. To achieve this, the visitor would explore the DAG held by the data structure, collecting the integer clock value, and a query based report from every accessible node. Having ensured that there are no duplicates, the reports are then sorted by the clock values and appended to create the final report of the history of the structure. Duplicate entries are easily identifiable since no report-clock value pair should be identical. Also, the fact that many events may have the same clock value is not a problem, as the construction of the DAG means that any two events with the same clock value are independent and so could have occurred in any order, or at the same time. As a result of this, and the general absence of data describing the execution order of independent events, the exact ordering of execution in the report may differ from that which was actually executed. However, all dependencies are guaranteed to occur in the correct order. A diagram demonstrating this process can be seen in Figure 5.

Given the amount of data stored, and that it has a much richer structure than just a list of events, it is possible to construct much more powerful tools for exploring the DAG's and providing output. These tools for example could be similar to those produced by the EU Provenance project [17], including query languages to allow the user to subset the provenance data produced by the individual functions. As the DAGs can be explored by visitors, it is possible to construct visitors which produce output that is compatible with existing tools for exploring filtering and mining data, not just tooling that has been produced for the benefit of provenance analysis.

### *F. Scaling and Fault Tolerance*

Currently this model builds and stores the DAGs in memory. This is possible because the number of operations performed on a give data structure is relatively small due to the nature of the analysis being performed. This results in the size of the DAGs remaining small in terms of memory, especially when

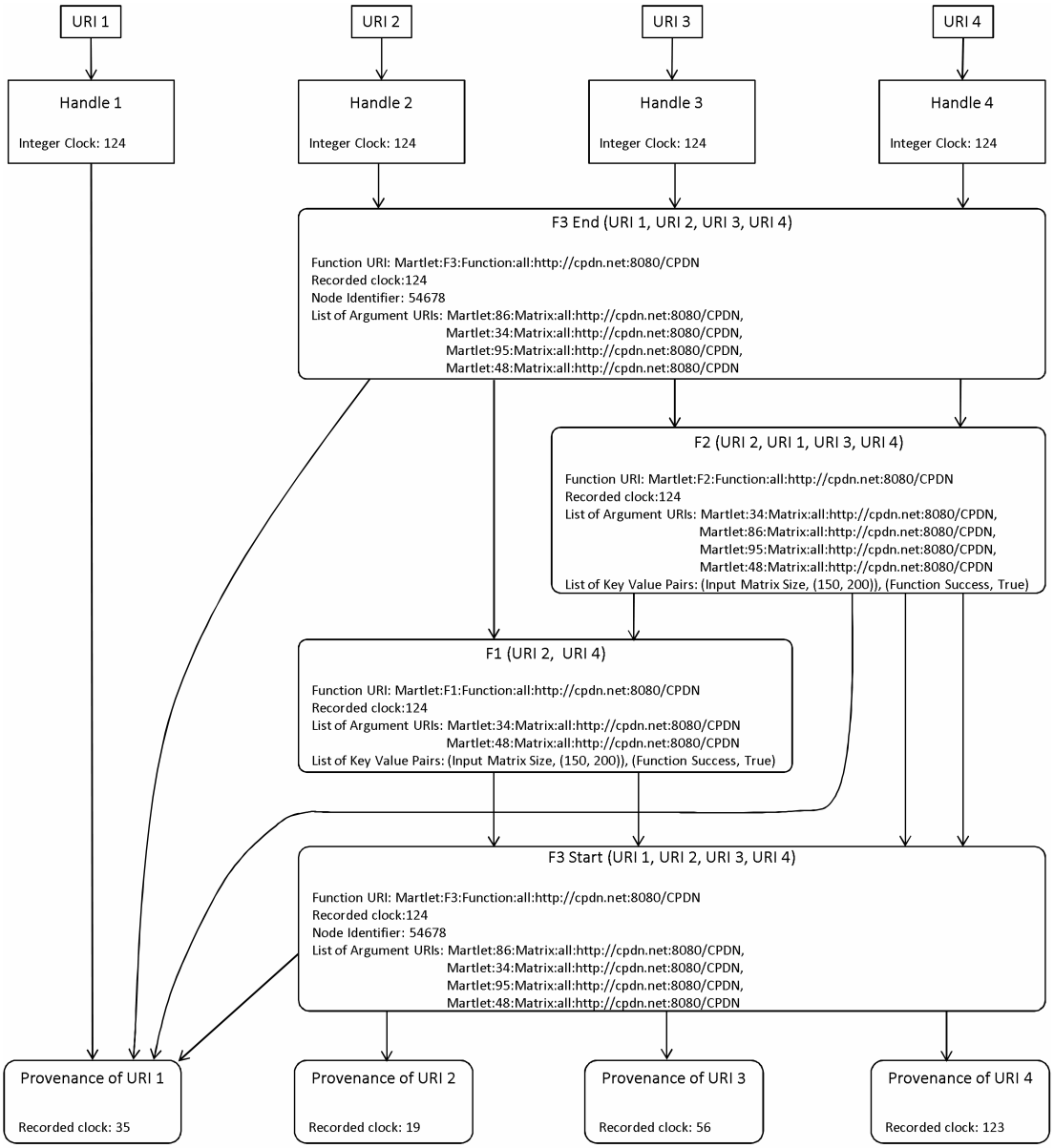



Fig. 4. A diagram representing the DAG produced from the state show in Figure 3a if `F` is a composite function made up of `F1` and `F2`, instead of being a base function. Within this composite, `F1` writes to URI 2 and `F2` writes to URI 3 and URI 4.

compared to the data structures they are describing, which can include very large matrices performing rough grained analysis by calling down to the Climate Data Analysis Toolkit [18]. Keeping the provenance in memory also means that in the event of the failure of a node, currently both the data and the provenance are lost. However, it is not reasonable to expect that such situations will always be, as Martlet may be deployed on systems using much finer grained analysis, and the middleware may choose or need to start recording data structures onto disk. This does not however mean that this model for keeping the provenance data has to be abandoned for models where the data structure size is smaller and less detailed. Instead it is just necessary to use a database on each server to store the nodes and vertices contained in the servers DAG instead of keeping the structure in memory. A combination of the clock values in the provenance nodes and the original URI of the data structures provide a unique key for storing each provenance node, since each URI is unique.

If it is also deemed that moving the provenance data with the data structures is too expensive, then when querying the provenance data it will be necessary to query several servers and the servers guarantee the availability of the provenance data. It is not realistic to require that databases must remain available indefinitely and all data entered into them must be stored indefinitely. An alternative would be to place an updateable lifetime on the provenance data entries. The middleware

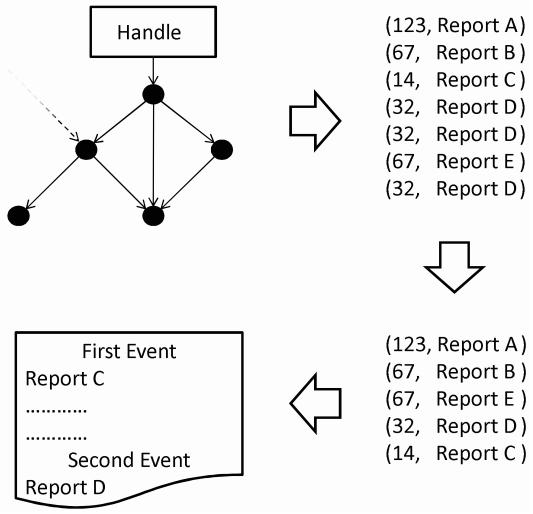



Fig. 5. A diagram showing the stages for the construction of a report from a DAG. First, reports are generated, then these are sorted and duplicates removed, before being used to construct the report. In reality, the removal of duplicates may occur at the same time as the pass over the DAG for efficiency reasons.

could then, for any data structures it holds, extend this at a predetermined interval, contacting the distributed data bases as required. This would then allow out of date data to be identified and removed. In the event of a server being down when there is a request for the provenance data, or when trying to update the life time, it will just be necessary to queue the request and try again later. It must be noted though, that this is a scale, trading the amount of meta data moved with the data structure against the amount of data transfer needed to maintain the middleware, so this position is only envisaged in for extreme cases where the provenance data is large and data structures are short lived.

## III. Conclusions

While there are already a number of provenance systems that take advantage of DAG's to store their data [8], this system is different from these other systems in that it is able to handle the two classes of data structure and the resulting complexity. This allows this work to provide insight into provenance systems for a different style of workflow language.

The provenance system itself is scalable, distributed, and, as discussed, able to adapt to both the changing structures of the workflows as they adapt to partitioned and distributed data, and the changing provenance data that functions generate during their execution. This is ability to store extra information is important in ensuring that the system can successfully interact with the wide range of legacy and external applications that users will seek to take advantage of. The provenance data can then be analysed using a range of tooling and techniques depending both on what is appropriate given the complexity of the functions the user has been executing, and any other tooling constraints the user may have.

## IV. Acknowledgments

This work is funded by the Natural Environmental Research Council and Microsoft. The author would like to thank Dr Andrew Martin and climate*prediction*.net for all their help.

## References

[1] D. Goodman and A. Martin, "Scientific middleware for abstracted parallelisation," Oxford University Computing Laboratory, Tech. Rep. RR-05-07, November 2005.

[2] D. Goodman, "Martlet; a scientific work-flow language for abstracted parallisation," in *Proceedings of the UK e-Science All Hands Meeting 2006*, S. J. Cox, Ed., National e-Science Centre. National e-Science Centre, September 2006.

[3] ——, "Introduction and Evaluation of Martlet, a Scientific Workflow Language for Abstracted Parallelisation," in *Proceedings of the 16th International World Wide Web Conference*, International World Wide Web Conference Committee. ACM, May 2007.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.

[5] P. Buneman, S. Khanna, and W. C. Tan, "Why and where: A characterization of data provenance." in *ICDT*, 2001, pp. 316–330.

[6] D. Stainforth, J. Kettleborough, A. Martin, A. Simpson, R. Gillis, A. Akkas, R. Gault, M. Collins, D. Gavaghan, and M. Allen, "Climate*prediction*.net: Design principles for public-resource modeling research," in *14th IASTED International Conference Parallel and Distributed Computing and Systems*, Nov 2002.

[7] C. Christensen, T. Aina, and D. Stainforth, "The challenge of volunteer computing with lengthy climate modelling simulations," in *Proceedings of the 1st IEEE Conference on e-Science and Grid Computing*, Melbourne, Australia, December 2005.

[8] L. Moreau, B. Ludäscher, I. Altintas, R. S. Barga, S. Bowers, S. Callahan, J. George Chin, B. Clifford, S. Cohen, S. Cohen-Boulakia, S. Davidson, E. Deelman, L. Digiampietri, I. Foster, J. Freire, J. Frew, J. Futrelle, T. Gibson, Y. Gil, C. Goble, J. Golbeck, P. Groth, D. A. Holland, S. Jiang, J. Kim, D. Koop, A. Krenek, T. McPhillips, G. Mehta, S. Miles, D. Metzger, S. Munroe, J. Myers, B. Plale, N. Podhorszki, V. Ratnakar, E. Santos, C. Scheidegger, K. Schuchardt, M. Seltzer, Y. L. Simmhan, C. Silva, P. Slaughter, E. Stephan, R. Stevens, D. Turi, H. Vo, M. Wilde, J. Zhao, and Y. Zhao, "Special issue: The first provenance challenge," *Concurr. Comput. : Pract. Exper.*, vol. 20, no. 5, pp. 409–418, 2008.

[9] R. Bird, *Introduction to Functional Programming using Haskell*, 2nd ed. Prentice Hall, 1998.

[10] *Apache Axis*, Apache Software Foundation, 2005, uRL: http://ws.apache.org/axis/.

[11] *The Apache Jakarta Project*, Apache Software Foundation, 2005, uRL: http://jakarta.apache.org/tomcat/.

[12] The MONET Consortium, "Monet architecture overview," The MONET Consortium, Tech. Rep., 2003, uRL: http://monet.nag.co.uk/cocoon/monet/.

[13] S. Parastatidis, J. Webber, P. Watson, and T. Rischbeck, "A grid application framework based on web services specifications and practices," North East Regional e-Science Centre, Tech. Rep., 2003. [Online]. Available: http://nersesc.ac.uk/projects/gaf

[14] K. Czajkowski, D. F. Ferguson, I. Foster, J. Frey, S. Graham, I. Sedukhin, D. Snelling, S. Tuecke, and W. Vambenepe, "The WS resource framework," Computer Associates International Inc and Fujitsu Limited, HP and IBM and The University of Chicago, Tech. Rep., 2004.

[15] W3C, *Simple Object Access Protocol (SOAP) 1.2*, 2003, uRL: http://www.w3c.org/TR/SOAP.

[16] P. Henderson and J. James H. Morris, "A lazy evaluator," in *POPL '76: Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages*. New York, NY, USA: ACM Press, 1976, pp. 95–103.

[17] L. Moreau, P. Groth, S. Miles, J. V. andJohn Ibbotson, S. Jiang, S. Munroe, O. Rana, A. Schreiber, V. Tan, and L. Varga, "The Provenance of Electronic Data," *Communications of the ACM*, Apr. 2008. [Online]. Available: http://www.ecs.soton.ac.uk/ lavm/papers/cacm08.pdf

[18] *Climate Data Analysis Tools*, British Atmospheric Data Centre, uRL: http://badc.nerc.ac.uk/help/software/cdat/.